Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Telemática

Proposta de especificação formal em SDL de uma rede de comunicação automotiva baseada no protocolo FlexRay com geração automática de código Java

Autor: Daniel Cesar Felisberto Rezende
Orientador: Walter da Cunha Borelli

Trabalho apresentado à Faculdade de Engenharia Elétrica e de Computação da UNICAMP como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica.

Comissão Examinadora

Prof. Dr. Walter da Cunha Borelli	**(Orientador)**
Prof. Dr. José Mario De Martino	**DCA/FEEC/UNICAMP**
Prof. Dr. Rafael Paoliello Guimarães	**Faculdade Salesiana de Vitória**

Campinas, 17 de Julho de 2009

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

R339p

Rezende, Daniel Cesar Felisberto
Proposta de especificação formal em SDL de uma rede de comunicação automotiva baseada no protocolo FlexRay com geração automática de código Java / Daniel Cesar Felisberto Rezende. --Campinas, SP: [s.n.], 2009.

Orientador: Walter da Cunha Borelli.
Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Java (Linguagem de programação de computador). 2. Redes de computação - Protocolos. 3. SDL (Linguagem de programação de computador). I. Borelli, Walter da Cunha. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Título em Inglês: A proposal for a formal specification using SDL of an In-vehicle network based on the FlexRay protocol with automatic Java code generation
Palavras-chave em Inglês: Java (Computer program language), Computer network protocols, SDL (Computer program language)
Área de concentração: Telecomunicações e Telemática
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: José Mario De Martino, Rafael Paoliello Guimarães
Data da defesa: 17/07/2009
Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidato: Daniel Cesar Felisberto Rezende

Data da Defesa: 17 de julho de 2009

Título da Tese: "Proposta de Especificação Formal em SDL de uma Rede de Comunicação Automotiva Baseada no Protocolo FlexRay com Geração Automática de Código Java"

Prof. Dr. Walter da Cunha Borelli (Presidente): _____

Prof. Dr. Rafael Paoliello Guimarães: _____

Prof. Dr. José Mario De Martino: _____

Agradecimentos

Agradeço primeiramente a Deus, pela saúde, pela sabedoria, pela paciência e pela constante proteção.

Aos meus pais, Paulo e Railda, que sempre me apoiaram e que mesmo nas horas difíceis não permitiram que eu desanimasse e assim seguisse em frente. O amor incondicional que eles me forneceram foi o maior combustível para a concretização deste trabalho.

Aos meus irmãos, Paulo e Tiago, e à minha irmã, Simone, pelas conversas diárias no skype, viagens divertidas nos momentos livres e pela união que transcende a ligação fraternal.

Ao meu orientador, Prof. Walter Borelli, pela oportunidade de pesquisa, pela orientação e pelas conversas descontraídas durante esse período.

Aos meus amigos da Bartira, que sempre forneceram entretenimento e diversão desde o primeiro momento em que cheguei a Campinas.

Finalmente, agradeço a todos meus familiares e amigos em geral, que em várias ocasiões me ajudaram a solucionar alguns problemas encontrados no caminho.

Dedico este trabalho

aos meus pais, Paulo e Railda;

aos meus irmãos Paulo Sérgio, Tiago e Simone;

e aos amigos que sempre me apoiaram

Resumo

Este trabalho apresenta uma proposta de especificação formal em SDL de uma rede de comunicação intraveicular baseada no protocolo de comunicação FlexRay com geração automática de código Java. O modelo proposto se baseia naquele apresentado na especificação padrão do protocolo FlexRay, porém algumas contribuições foram feitas a fim de gerar uma rede FlexRay funcional e executável. O modelo SDL gerado confere uma formalização comportamental ao sistema, permitindo a sua validação e a simulação de suas principais funcionalidades e casos críticos através da ferramenta SDL TAU *Suite*. Depois de o sistema ser modelado, simulado e validado é gerado código Java para execução do sistema criado. Para isso, foi desenvolvida uma ferramenta geradora de código Java que recebe como entrada um arquivo com as especificações em SDL e tem-se como resultado um sistema descrito em Java que faz uso de *sockets* para comunicação entre os processos.

Palavras-Chave: Protocolos de Comunicação, Sistemas Intraveiculares, FlexRay, Especificação Formal, SDL - Specification and Description Language, SDT - SDL TAU Suite, MSC – Message Sequence Chart, Simulação, Validação, Java, Geração de Código.

Abstract

This work presents a proposal of formal specification using SDL for an in-vehicle network based on the FlexRay protocol with automatic generation of Java code. The proposed model is based on that presented in the standard specification of the FlexRay protocol, although some contributions were made in order to generate a functional and executable FlexRay network. The SDL model generated provides the system a behavioural formalization, making it possible to validate and simulate its key features and critical cases by the use of the tool TAU SDL Suite. After the system is modeled, simulated and validated is generated Java code for implementing the system created. For this reason it was developed a tool for generating Java code that receives as input a file with the specifications in SDL, and delivers as result a system written in Java that makes use of sockets for communication between processes.

Keywords: Communications Protocols, In-Vehicle Systems, FlexRay, Formal Specification, SDL - Specification and Description Language, SDT - SDL TAU Suite, MSC – Message Sequence Chart, Simulation, Validation, Java, Code Generation.

Sumário

Lista de Figuras

Lista de Tabelas

Acrônimos e Siglas

ASN.1 - Abstract Syntax Notation One

BNF - Back-Naur Form

CAN - Controller Area Network

CASE - Computer Aided Software Engineering

CCITT - Comité Consultatif International Télégraphique et Telephonique

CE - Communication Element

CHI - Controller Host Interface

CHILL - CCITT High Level Language

CODEC - Coding and Decoding Process

CRC - Cyclic Redundancy Check

CSP - Clock Synchronization Process

CSS - Clock Synchronization Startup Process

ECU - Electronic Control Unit

ERRN - Error Not signal

FSP - Frame and Symbol Processing

FTDMA - Flexible Time Division Multiple Access

IDL - Interface Description Language

IP - Internet Protocol

ITU-T - International Telecommunications Union - Telecommunication

JavaCC - Java Compiler Compiler

JDK - Java Development Kit

LIN - Local Interconnect Network

MAC - Media Access Control Process

MOST - Media Oriented Systems Transport

MSC - Message Sequence Chart

MT - Macrotick

MTG - Macrotick Generation Process

NIT - Network Idle Time

Pid - Process Identification

POC - Protocol Operation Control

SAE - Society of Automotive Engineers

SDL - Specification and Description Language

SDT - SDL Design Tool

ST - Sampletick

STBN - Standby Not signal

TCP - Transmission Control Protocol

TDMA - Time Division Multiple Access

TTCN - Tree and Tabular Combined Notation

TTP - Time-Triggered Protocol

UML - Unifying Modeling Language

WUP - Wakeup Pattern

µT - Microtick

Trabalho Publicado pelo Autor

D. Rezende, W. Borelli. Proposta de uma Rede de Comunicação Automotiva Baseada no FlexRay. In: XXV Simpósio Brasileiro de Telecomunicações (SBrT'07), Recife, PE, Setembro 2007

Capítulo 1

Introdução

1.1. Motivação e Objetivos

Recentemente tem havido um aumento significativo na quantidade de eletrônica introduzida no carro, e é esperado que esta tendência continue enquanto as montadoras introduzirem avanços adicionais em segurança, confiabilidade e conforto. Os veículos atuais são equipados com um grande conjunto de sensores e atuadores e cada aplicação consiste de uma ou mais Unidades de Controle Eletrônico (do termo em inglês, *Electronic Control Unit* - ECU). Em um sistema automotivo consistindo de diversas aplicações mais de 70 ECUs podem distribuir mais de 2500 sinais de dados [7]. Integrar as informações destes sensores e atuadores não é uma tarefa fácil, já que muitas vezes é exigido um meio físico construído com uma fiação enorme, isto sem contar o número de conectores.

Motivados pelas aplicações intraveiculares, várias companhias vêm investindo no projeto de controladores que pudessem gerenciar o tráfego de informações com interfaces através de um meio físico reduzido, geralmente um barramento serial, porém capaz de possibilitar a multiplexação dessas informações. A esta forma de conexão é dado o nome de Rede Intraveicular (*In-Vehicle Networking*).

O uso de uma rede intraveicular oferece muitos benefícios, dentre eles: requer uma quantidade bem menor de fios e conectores, diminuindo custos materiais e de instalação; possibilita o compartilhamento de sensores e de outras medições disponíveis na rede; diminui o tempo de manufatura do veículo (exatamente pela menor quantidade de cabeamento necessário); flexibiliza o projeto, já que um novo sistema pode ser projetado apenas readaptando o software de controle e ainda possibilita a modularização do projeto do sistema e da execução dos testes de validação, aumentando a confiabilidade da implementação e reduzindo os prazos envolvidos no desenvolvimento.

A aplicação das redes intraveiculares exigiu que fossem criadas regras pré-definidas para o gerenciamento da comunicação e, em particular, para conceder acesso ao barramento por parte dos nós. A estas regras pré-definidas dá-se o nome de protocolo de comunicação automotivo. Protocolos de comunicação automotivos são meios de transmissão e recepção de dados utilizados para intercomunicarem módulos eletrônicos e/ou sensores e atuadores inteligentes equipados com microcontroladores e *transceivers*, por exemplo. Existem vários tipos de protocolos de comunicação, cada qual com suas características técnicas específicas e, portanto, com as suas aplicações mais apropriadas. Conforme a SAE (*Society of Automotive Engineers*) [10], existem três classes de protocolos de comunicação:

- Classe A – até 10kbps (kbps = 1000 bits por segundo)
- Classe B – até 125kbps
- Classe C – acima de 125kbps

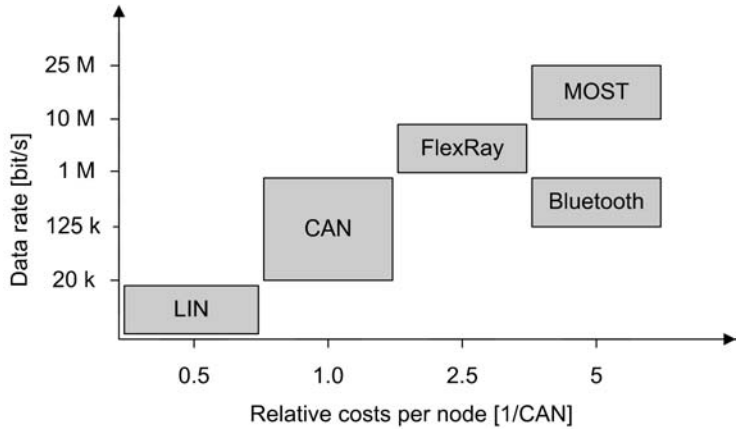


Figura 1.1: Taxa de dados e custo relativo dos sistemas automotivos [5].

Na classe A estão os protocolos com taxa de transmissão de até 10kbps e que geralmente relacionam-se às funções de conforto de um veículo. Na classe B estão os protocolos com taxa de transmissão que vão de 10kbps até 125kbps e que geralmente relacionam-se ao controle dos sistemas de entretenimento de um veículo. E na classe C estão os protocolos que possuem taxa de transmissão acima de 125kbps e que geralmente relacionam-se ao controle dos sistemas de segurança de um veículo.

Em um sistema automotivo moderno há a necessidade de tráfego de dados orientados por tempo (*time-triggered*) e orientados por eventos (*event-triggered*). Em sistemas orientados por tempo todas as comunicações e atividades de processos são iniciadas em instantes pré-determinados, enquanto que em sistemas orientados por eventos todas as comunicações e

atividades de processos são iniciadas quando existem mudanças nas variáveis de estado, ou seja, chegada de um evento. O tráfego orientado por eventos é mais adequado a aplicações não-críticas (*soft real-time*) devido a sua maior flexibilidade e eficiência de recursos. Já o tráfego orientado por tempo é mais adequado a aplicações críticas (*hard real-time*) devido a seu melhor desempenho no gerenciamento de modelos tolerante a falhas [6].

Atualmente os sistemas automotivos incorporam funções perceptíveis ao usuário final, tais como sistemas de ar-condicionado automático, navegação inteligente, sistemas integrados de multimídia, controle de frota e funções que facilitam o diagnóstico e manutenção. Para estes tipos de sistemas não-críticos um número de protocolos tornou-se popular, tais como LIN (*Local Interconnect Network*) [35], CAN (*Controller Area Network*) [36] e MOST (*Media Oriented Systems Transport*) [37]. Para aplicações críticas e mais complexas (aplicações *X-by-Wire* e aplicações relacionadas à transmissão e motor) o FlexRay surge como protocolo mais adequado a operações em tempo real [3].

O protocolo LIN foi desenvolvido para ser uma sub-rede de baixa taxa de transmissão e de baixo custo que interligasse os sensores inteligentes e os atuadores. O MOST é uma rede de comunicação sobre uma topologia de anel de fibra-óptica utilizada na interligação de componentes multimídia internamente nos veículos.

O protocolo CAN ainda é o protocolo mais usado em redes intraveiculares e pode ser encontrado em várias aplicações em um veículo. Isso devido a sua grande capacidade de manejo de tráfego orientado a eventos. A transmissão de dados na rede CAN alcança taxas de até 1 Mbps, embora 500Kbps seja a escolha mais comum para, por exemplo, controle do motor e sistemas ABS [1]. Porém, o CAN carece de mecanismos de tolerância a falhas e de capacidade de largura de banda. Conseqüentemente, o CAN não é indicado para soluções de *X-By-Wire*.

Outro protocolo que concorre com o protocolo FlexRay é o TTP (*Time-Triggered Protocol*) [38]. O TTP foi desenvolvido para redes altamente tolerante a falhas e dirigido para sistemas críticos tais como *X-By-Wire* e sistemas aviônicos. Um ponto fraco deste protocolo é que devido a sua inflexível transmissão de mensagem e alto custo, o uso de outro protocolo de comunicação seria necessário para outras aplicações no carro, como *Powertrain* (sistema de transmissão e motor), onde alta largura da banda juntamente com capacidade de transmissão orientada a eventos é imprescindível. Enquanto FlexRay suporta diretamente transmissão orientada a eventos, TTP não suporta. E como FlexRay combina transmissão de mensagens TDMA (Time

Division Multiple Access) e FTDMA (Flexible Time Division Multiple Access), pode permitir transmissão de mensagens orientadas por tempo e por eventos. Conseqüentemente, usando FlexRay é possível desenvolver sistemas *X-By-Wire* e sistemas *Powertrain*, reduzindo assim a necessidade por várias tecnologias de protocolos de comunicação.

Entre os protocolos mais promissores o FlexRay tem o maior potencial para se tornar padrão de rede intraveicular para aplicações críticas e tolerantes a falhas, principalmente porque é fortemente apoiado por sócios industriais e incentivado pela maioria dos principais fabricantes de automóveis dos quais vários migraram de situação de apoiadores do TTP para serem apoiadores do FlexRay [13]. Embora FlexRay seja desenvolvido e principalmente planejado para comunicações de segurança-crítica como *X-By-Wire*, é possível usá-lo em várias outras aplicações automotivas.

A introdução de sistemas avançados de controle que combinam sensores múltiplos, atuadores e ECUs está começando a demandar exigências na tecnologia de uma comunicação não encontrada atualmente nos protocolos de comunicação existentes. As exigências adicionais para as aplicações futuras do controle intraveicular incluem:

- Combinação de taxas de dados mais elevadas.
- Comportamento determinístico (previsível e planejado).
- Sustentação da tolerância a falhas.

A introdução desses sistemas *X-by-Wire* será um passo significante para a indústria automobilística [4]. O termo *X-by-Wire* é utilizado quando um sistema eletromecânico, formado por um módulo de controle, sensores e atuadores, substitui um sistema puramente mecânico. A parcela *X* deste termo dá lugar a qualquer sistema de segurança existente em um veículo, como, por exemplo, *Steer-by-Wire* (sistema de controle de tração e estabilidade), *Brake-by-Wire* (sistema de freios) e *Suspension-by-Wire* (sistema de suspensão), entre outros. Os sistemas *by-Wire* devem ser capazes de sentir e tratar apropriadamente uma solicitação do motorista e tomar as ações necessárias, considerando as circunstâncias presentes, de dirigibilidade e de ambiente. Algumas empresas automotivas ainda demonstram hesitação na implementação dos sistemas *X-by-Wire*, mesmo porque os sistemas mecânicos têm provado, ao longo do tempo, ser extremamente confiáveis. Por outro lado, ano após ano, temos notado o aumento do número de sistemas de segurança utilizados nos veículos de passageiros. Essa evolução demanda o

aprimoramento das tecnologias utilizadas, buscando facilitar a montagem e a manutenção dos sistemas e a otimização do desempenho esperado pelos clientes.

Como o protocolo FlexRay tem um futuro muito promissor e provavelmente se tornará um padrão universal em sua categoria, várias empresas tem investido em pesquisa e desenvolvimento de sistemas para esse protocolo. Porém, as ferramentas encontradas para desenvolvimento e implementação de aplicações automotivas usando o FlexRay são comerciais (tais como: [33] e [34]) e, portanto, não dão acesso ao funcionamento interno desse protocolo. Isso faz com que haja uma limitação da sua evolução, evitando assim possíveis melhorias e extensões desse protocolo por parte da comunidade científica e/ou industrial. A própria especificação padrão do protocolo FlexRay [5] não descreve completamente o comportamento dos mecanismos do sistema (descrição feita por meio da linguagem de especificação formal SDL). Nesse sentido, a intenção da especificação do protocolo não era descrever o completo funcionamento do sistema, e sim apenas apresentar de forma superficial os mecanismos do núcleo do protocolo e suas interações. Em muitos casos o método de descrição foi escolhido de modo a facilitar o entendimento do usuário e não visando a eficiência da implementação.

Motivada pela limitação de desenvolvimento oferecida pelas ferramentas comerciais e pela superficialidade na descrição do comportamento dos mecanismos do protocolo encontrada em [5], esta dissertação propõe uma implementação de uma rede FlexRay através de um modelo SDL funcional e executável. Junto com a modelagem da rede FlexRay em SDL também foi desenvolvido um compilador de SDL para Java como uma forma de poder executar o sistema modelado e oferecer ao projetista uma interação com o sistema. O modelo proposto se baseia naquele apresentado na especificação padrão do protocolo FlexRay, porém algumas contribuições foram feitas com o objetivo de aprimorar a eficiência de alguns dos mecanismos do protocolo, melhorando o resultado final do sistema que venha a ser desenvolvido. Para criar uma rede FlexRay completa e funcional sua estrutura teve que ser projetada, já que estes detalhes não são descritos na especificação do protocolo. Dessa forma, a arquitetura do sistema, a topologia da rede, a integração entre os principais mecanismos dentro do controlador de comunicação e a interface que conecta o controlador de comunicação ao canal de comunicação foram todos criados e descritos usando a linguagem SDL.

A metodologia proposta consiste na modelagem da rede FlexRay utilizando a especificação formal em SDL e de geração de código em Java. O modelo SDL gerado confere uma

formalização comportamental bastante interessante ao sistema, permitindo a sua validação, bem como a simulação de suas principais funcionalidades e casos críticos, o que auxilia na identificação e na correção de quaisquer erros existentes antes de uma eventual implementação.

Depois de especificar a rede FlexRay proposta, as funcionalidades desejadas foram completamente testadas. O sistema foi integralmente validado através da ferramenta *Validator* disponível no pacote SDT (SDL Design Tool) [12, 15] da *Telelogic*. Após a completa validação do sistema, foram simuladas as principais funcionalidades do sistema FlexRay para a garantia da validade do sistema. Para tanto foi utilizada a ferramenta *Simulator*, que também é parte integrante do pacote SDT. Depois de o sistema ser modelado, simulado e validado é gerado código Java para execução do sistema criado. Para isso, foi desenvolvida uma ferramenta geradora de código Java que recebe como entrada qualquer arquivo com especificações em SDL e tem-se como resultado um sistema descrito em Java que faz uso de sockets para comunicação entre os processos.

A linguagem Java foi escolhida por várias razões, porém duas delas se destacam: a independência de plataformas [25] e a facilidade de criação de interfaces gráficas. Estas características trazem não só portabilidade, mas também uma melhor interação do projetista com o sistema gerado. Para a construção da ferramenta geradora de código Java foi utilizada a ferramenta construtora de compiladores *JavaCC* [16] (*Java Compiler Compiler*), que pode ser obtida gratuitamente na Internet.

A metodologia proposta visa auxiliar na prototipagem de aplicações automotivas, antes mesmo de serem implementadas, pois permite ao projetista de aplicações automotivas ter acesso completo a todos os níveis de desenvolvimento do sistema. Ao contrário do que acontece com algumas ferramentas comerciais que oferecem soluções de implementação de aplicações automotivas usando o FlexRay [33, 34], porém não dão acesso ao funcionamento interno do protocolo.

1.2. Organização da Dissertação

Este trabalho está dividido em seis capítulos e três apêndices.

O Capítulo 2 descreve uma introdução ao protocolo de comunicação automotiva FlexRay, considerando os pontos mais relevantes para o trabalho desenvolvido.

O Capítulo 3 apresenta uma proposta de especificação formal de uma rede de comunicação automotiva baseado no FlexRay. No capítulo é apresentado o modelo de sistema desenvolvido através da descrição da estrutura projetada e da descrição dos diagramas em SDL dos principais mecanismos do protocolo FlexRay. Também são apresentadas algumas adaptações feitas na especificação do protocolo a fim de possibilitar a elaboração desta rede de comunicação usando o protocolo FlexRay

No Capítulo 4 são apresentados os resultados da validação do sistema desenvolvido e exemplos de simulações das principais funcionalidades e casos críticos através de diagramas MSC (*Message Sequence Chart*). O capítulo apresenta também técnicas para identificar e corrigir erros existentes de lógica e especificação destes sistemas através da validação e simulação, garantindo maior sucesso em futuras expansões e implementações.

O capítulo 5 apresenta a ferramenta desenvolvida para a geração de código Java a partir das especificações SDL e os principais conceitos envolvidos. São apresentados também exemplos de construções em SDL e seus equivalentes em Java.

No Capítulo 6 são apresentadas as conclusões finais e sugestões para eventuais trabalhos futuros.

O Apêndice A apresenta uma introdução à linguagem SDL com as características principais e relevantes para o trabalho desenvolvido. O apêndice também apresenta a ferramenta SDT (*SDL TAU Suite)* que possui os módulos *Simulator* e *Validator* utilizados para gerar resultados das simulações dos casos críticos e validação dos sistemas respectivamente, para o desenvolvimento das especificações dos sistemas em SDL.

No Apêndice B são introduzidos conceitos de interpretação de código utilizados na construção da ferramenta geradora de código através do auxílio da ferramenta *JavaCC*.

O Apêndice C apresenta um artigo [13] publicado no período.

Capítulo 2

FlexRay e a Metodologia Utilizada

Neste capítulo são introduzidos conceitos relacionados ao sistema FlexRay e é apresentada a metodologia proposta para o desenvolvimento da rede FlexRay. Na seção 2.1 é apresentada uma introdução do protocolo FlexRay, mostrando o seu funcionamento e suas principais características. Finalizando o capítulo, na seção 2.2 é proposta a metodologia para o desenvolvimento da rede FlexRay através do uso combinado da linguagem de especificação formal SDL e da geração automática de código Java.

2.1. Protocolo de Comunicação FlexRay

O protocolo FlexRay surgiu em 1999 como resultado da cooperação entre BMW e DaimlerChrysler depois que as duas montadoras automotivas perceberam que as soluções atuais não satisfariam suas necessidades para aplicações futuras incluindo *X-By-Wire*. Como resposta a isso, o Consórcio FlexRay [1] foi formado com o objetivo de desenvolver um novo protocolo, chamado FlexRay [4] e [5]. Este novo protocolo deveria ser a solução não só para a introdução dos sistemas *X-By-Wire*, mas também para a substituição de alguns protocolos adotados atualmente, assim reduzindo o número total de redes intraveiculares [6].

O protocolo FlexRay é uma combinação do protocolo Byteflight da BMW e do protocolo TTP da Universidade Técnica de Vienna. O Byteflight foi originalmente projetado para sistemas de segurança passivos em aplicações como acionamento de *airbag*, onde curtos tempos de resposta são necessários. Visto que o tempo de ação de um sistema de *Airbag* é curto (na ordem de milisegundos), a probabilidade de ocorrer uma falha em um espaço tão curto de tempo é baixa. Consequentemente Byteflight não precisa suportar tolerância a falhas, e dessa forma, não é adequado para sistemas de controle ativos, como *X-By-Wire*, que tem um tempo de ação longo e, portanto, requer tolerância a falhas [18].

O sistema FlexRay é mais do que um protocolo de comunicação, inclui também especificações para transmissões de alta velocidade e definições de interfaces de hardware e software, entre os componentes de um nó FlexRay.

2.1.1. Topologias de Rede

O FlexRay utiliza várias ECUs, mas apenas uma comunica de cada vez. Isto quer dizer que este protocolo vai ter, pelo menos, um ECU emissor e um ECU receptor, os quais comunicam entre si através de um barramento. A topologia de grupo de comunicação do protocolo FlexRay permite ter um canal (*Single Channel*) ou dois canais (*Dual Channel*). Cada grupo de comunicação pode ter uma topologia em barramento, estrela e híbrida.

Na topologia de barramento, cada uma das ECUs pode ser ligada diretamente a um ou a dois canais.

Numa topologia em forma de estrela, os pontos centrais de ligação são denominados acopladores de estrela, e caso seja usada esta topologia é possível construir redes em cascata ligando assim dois acopladores de estrela diretamente.

Por último, a topologia híbrida, resulta de uma combinação de uma topologia de barramento e uma topologia estrela.

2.1.2. Estrutura de um Nó FlexRay

A arquitetura do nó de um sistema FlexRay é composta de um computador principal (*host*), de um controlador de comunicação (*communication controller*) e de um controlador de barramento (*bus driver*).

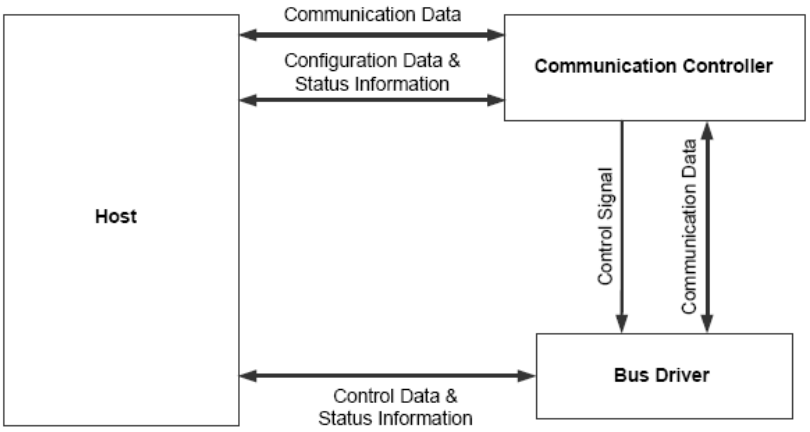


Figura 2.1: Arquitetura de um Nó FlexRay [5]

O host inicializa todos os componentes do nó (controladores de comunicação e de barramento) e os controla durante a execução da rede. O host fornece ao controlador de comunicação informações de controle e configuração e os dados a serem transmitidos durante o ciclo de comunicação. O controlador de comunicação, por sua vez, fornece ao host informações sobre o estado atual dos componentes do núcleo do protocolo e também lhe entrega os dados recebidos das mensagens.

Cada canal de comunicação tem um controlador de barramento para conectar o nó ao canal. A comunicação feita entre o controlador de barramento e o controlador de comunicação refere-se a sinais de controle e dados a serem transmitidos ou dados recebidos através do canal. Existe ainda uma comunicação entre o controlador de barramento e o host que permite ao último controlar os modos de operação do controlador de barramento, ler as condições de erro e obter informações do estado atual do controlador de barramento.

2.1.3. Processo de Comunicação

No protocolo FlexRay, o controle de acesso ao meio físico é baseado em um ciclo de comunicação recorrente (Figura 2.2). O ciclo de comunicação consiste de um segmento estático, um segmento dinâmico (configurável), uma janela de símbolos (opcional) e, finalmente, uma fase em que a rede está em modo de espera, o chamado NIT (*Network Idle Time*).

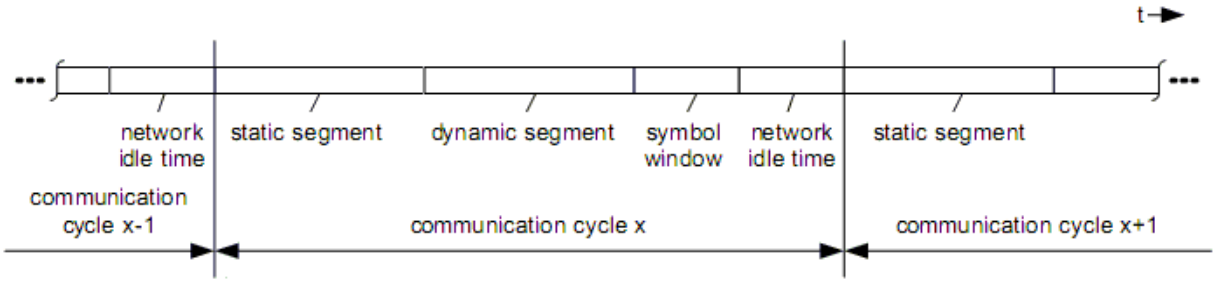


Figura 2.2: Ciclo de comunicação do FlexRay [18]

A fronteira entre os segmentos estáticos e dinâmicos pode ser livremente definida pelo projetista antes da execução do sistema, sendo que um segmento pode até não existir. O ciclo de comunicação começa com um símbolo de sincronismo. Os slots de tempo são identificados por números de identificação que são usados tanto para os segmentos estáticos quanto para os dinâmicos. Todos os nós da rede realizam sincronização do clock. No entanto, apenas as mensagens selecionadas do segmento estático são usadas para sincronização do clock.

O segmento estático do ciclo de comunicação é composto de um número de slots de transmissão livremente configuráveis e de comprimento idêntico. O segmento estático está situado no início do ciclo de comunicação. O identificador da mensagem corresponde ao respectivo slot de tempo em que a mensagem será transmitida. Se uma mensagem não é transmitida (por exemplo, se um nó falhar), então o tempo corre sem ser utilizado. O acesso ao meio de comunicação ocorre utilizando o princípio da TDMA, onde as mensagens têm alocação em slots de tempo fixos na qual eles têm acesso exclusivo ao meio. Cada nó conectado à rede pode transmitir uma ou mais mensagens em slots reservados para ele. Os slots de tempo são repetidos em um ciclo fixo. O momento em que uma mensagem está no barramento pode ser exatamente previsto e o acesso ao meio é, portanto, determinístico.

No entanto, a atribuição de fixar a largura de banda para os componentes ou mensagens por meio de fixos slots de tempo tem a desvantagem de que a banda não é totalmente aproveitada. Por esta razão o FlexRay subdivide o ciclo em um segmento estático e um segmento dinâmico.

No segmento dinâmico, a transmissão ocorre em conformidade com a especificação *byteflight* [18]. Os contadores de slots em todos os nós contam de forma síncrona e crescente. O acesso ao barramento segue o método de mini-slotting, também conhecido como FTDMA. No segmento dinâmico, o acesso é permitido dinamicamente de acordo com o nível de prioridade atribuída aos nós que tenham dados para transmitir, através do valor de um identificador específico contido no cabeçalho da mensagem. O objetivo deste segmento é proporcionar comunicação dentro dos limites do tempo e da largura de banda. Cada um dos nós é livre para variar a duração dos slots (sujeito ao controle do projetista do sistema) de acordo com a quantidade de dados que deseja transmitir.

O ciclo de comunicação ainda possui um segmento opcional que é a janela de símbolos. A janela de símbolos é um período de comunicação em que um símbolo pode ser transmitido na rede. O objetivo da janela de símbolos é enviar símbolos especiais, por exemplo, símbolo de ativação dos nós na rede e símbolo de frame nulo para inicialização da rede. Dentro da janela de símbolo um único símbolo pode ser enviado.

E o último segmento do ciclo é o NIT (Network Idle Time). Este segmento representa o período sem comunicação dentro da rede. É durante esta fase que o nó calcula e aplica a correção do seu clock.

2.1.4. Formato da Mensagem

A mensagem consiste de três segmentos: cabeçalho (header), campo de dados (payload) e o campo do final da mensagem (trailer).

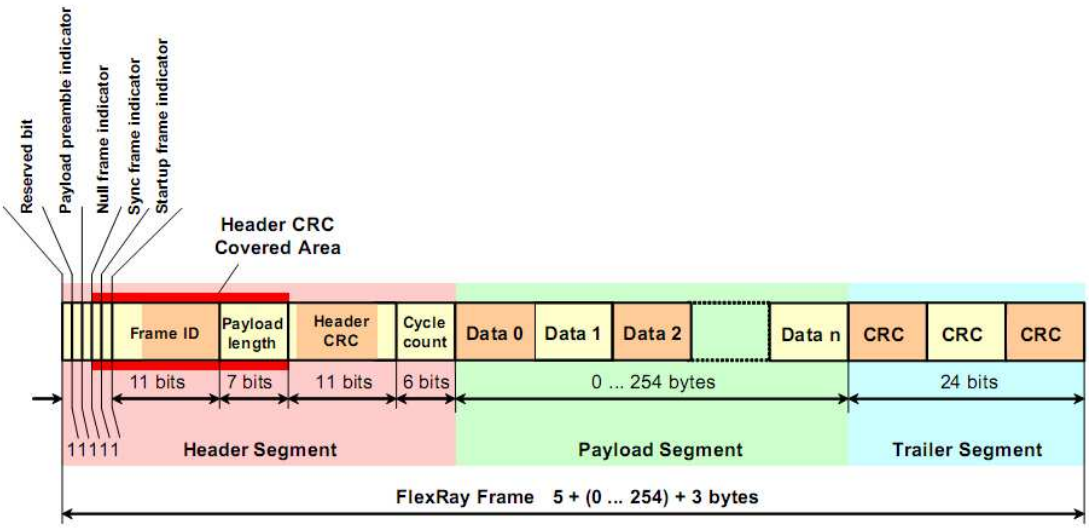


Figura 2.3: Formato da mensagem do protocolo FlexRay [18]

O nó deve transmitir a mensagem na rede tal que o segmento do cabeçalho apareça em primeiro lugar, seguido pelo segmento de dados e, em seguida, o segmento do final da mensagem. O nó enviará os campos seguindo a ordem da esquerda para a direita como demonstrado na Figura 2.2.

O segmento do cabeçalho consiste de 5 bytes. Esses bytes subdividem-se em bit reservado, indicador de dados, indicador de mensagem nula, indicador de mensagem de sincronismo, indicador de mensagem de inicialização, identificador da mensagem, tamanho dos dados, CRC (*cyclic redundancy check*) do cabeçalho e o contador de ciclo. O identificador da mensagem é o campo mais importante do cabeçalho, pois ele define o slot em que a mensagem deve ser transmitida no segmento estático e define o nível de prioridade no segmento dinâmico. O identificador de mensagem não é usado mais do que uma vez durante o ciclo de comunicação. Cada mensagem que deve ser transmitida em uma rede tem um identificador de mensagem atribuído a ela.

O segmento de dados contém de 0 a 254 bytes. Os bytes do segmento de dados são identificados numericamente, começando em 0 para o primeiro byte depois do segmento do cabeçalho e incrementado em 1 para cada byte seguinte.

O segmento do final da mensagem (*trailer*) contém um único campo composto de 24 bits de CRC da mensagem. Esse campo contém o código CRC calculado sobre os campos do cabeçalho e de dados. Nesse cálculo estão inclusos todos os campos desses segmentos.

2.1.5. Representação do tempo no protocolo FlexRay

A representação do tempo no protocolo FlexRay é baseada em uma hierarquia que inclui *microticks* e *macroticks*, como mostra a Figura 2.3.

O ciclo de comunicação consiste de um número inteiro de *macroticks*. O número de *macroticks* por ciclo é idêntico para todos os nós de um único grupo (cluster) e continua a ser o mesmo de um ciclo para outro. Todos os nós devem estar no mesmo número de ciclo em cada instante.

Microticks são unidades de tempo derivadas diretamente do oscilador do microcontrolador (clock interno). Cada nó, então, cria seus próprios *microticks* cuja estrutura é tal que não pode ser afetada por qualquer mecanismo de sincronização. Assim, todos os nós têm diferentes valores de *microticks* que representam a granularidade específica de cada nó. Eles serão usados para criar os *macroticks* e, em conjunto com estes, realizarão a fase de sincronização entre *microticks* e *macroticks*.

O *macrotick* é um intervalo de tempo, relativo a um conjunto específico de participantes na rede, calculado por uma rotina de sincronização algorítmica. Portanto, não é criado por um simples mecanismo eletrônico, como o *microtick*, mas é o resultado de um cálculo qualificado. De fato, o *macrotick* é a menor unidade da granularidade de tempo global da rede. A duração local de cada *macrotick* de um nó é inevitavelmente constituída por um número inteiro de *microticks*, este número a ser calculado e ajustado pelo algoritmo de sincronização do clock. Como o valor do *microtick* é específico para cada nó, como já mencionado, sendo dependente da freqüência do seu oscilador local, o número de *microticks* por *macrotick* também pode variar entre nós.

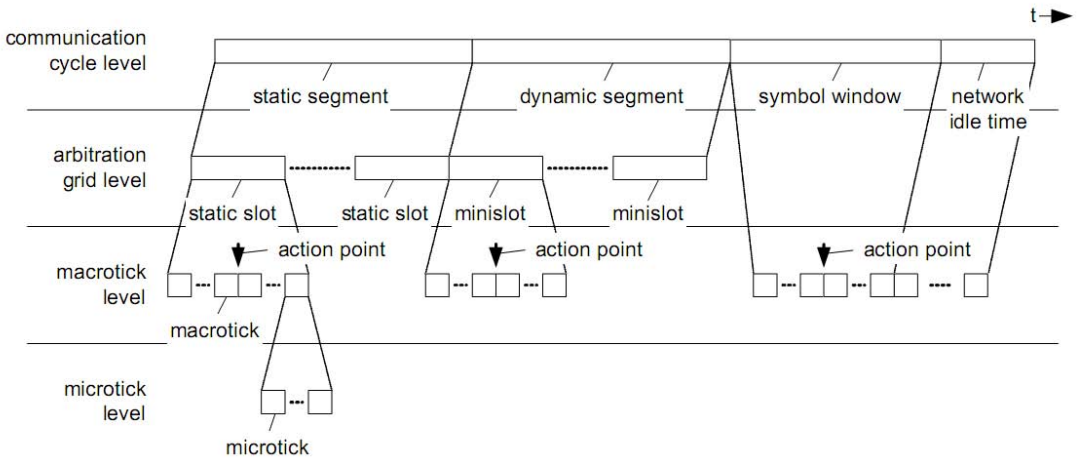


Figura 2.4: Hierarquia de tempo dentro do ciclo de comunicação [5]

2.2. Metodologia para o desenvolvimento da rede FlexRay

Este trabalho propõe a utilização de uma metodologia que combina as técnicas de especificação formal em SDL com a geração de código em Java para execução do sistema desenvolvido.

A idéia básica desta metodologia (Figura 2.5) é utilizar a técnica de especificação formal em SDL em uma fase inicial do desenvolvimento visando prover uma formalização comportamental do sistema FlexRay (capítulo 3). O modelo SDL gerado permite que seja feita a validação, bem como a simulação de suas principais funcionalidades e casos críticos, o que auxilia na identificação e na correção de quaisquer erros existentes antes de uma eventual implementação (capítulo 4).

Em uma fase posterior, é gerado código Java a partir das especificações SDL desenvolvidas (capítulo 5).

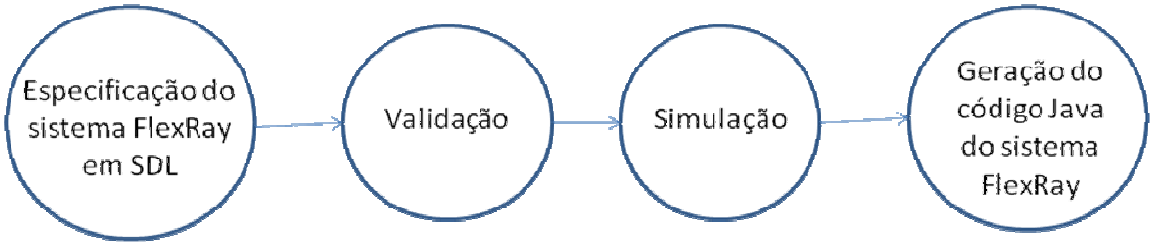


Figura 2.5: Metodologia combinando o uso de SDL e geração de código Java

Para o processo de geração automática de código, foi desenvolvida uma ferramenta para a conversão das especificações em SDL para código em Java com o auxílio do JavaCC (*Java*

Compiler Compiler) [16], que é um programa que ajuda na construção de ferramentas interpretadoras de código, ou seja, de compiladores.

Com o uso desta metodologia têm-se grandes vantagens decorrentes do uso da linguagem SDL (formalização comportamental, possibilidade de validação e simulação do sistema) e da geração automática de código em Java (agilização do processo de implementação, redução na incidência de erros devido à automatização do processo e portabilidade do sistema devido ao uso da linguagem Java).

Para a automação desta metodologia, utilizou-se o SDT (SDL Design Tool), que é uma ferramenta CASE (*Computer Aided Software Engineering*) composta por módulos responsáveis por:

- especificar sistemas SDL;
- simular, verificar e validar as especificações;
- editar e gerar diagramas MSC como resultado de simulações.

O uso do ambiente SDT (apresentado no apêndice A) e do JavaCC (apresentado no apêndice B) tornaram o trabalho de especificação e de geração automática de código, respectivamente, atividades mais simples e eficientes.Nos próximos capítulos serão utilizados termos técnicos, palavras de língua estrangeira ou nomes próprios referentes à modelagem do sistema. Para facilitar o entendimento do leitor foi adotada uma padronização no estilo da fonte específico para cada tipo de termo utilizado. Assim, será utilizado o estilo *itálico* para o uso de palavras e termos técnicos de origem estrangeira e será adotado o estilo ***negrito itálico*** para nomes próprios referentes à modelagem do sistema.

Capítulo 3

Especificação em SDL de uma Rede de Comunicação FlexRay

Com base nas descrições apresentadas no capítulo 2, referentes à estrutura do protocolo de comunicação FlexRay, neste capítulo é apresentada a proposta de especificação formal em SDL (*Specification and Description Language*) de uma rede de comunicação automotiva funcional e executável utilizando o protocolo FlexRay. Esta proposta de modelagem da rede FlexRay difere da modelagem encontrada na especificação original do protocolo FlexRay [5], pois esta apenas descreve o comportamento dos mecanismos do controlador de comunicação do protocolo FlexRay e omite a forma como a rede FlexRay pode ser implementada.

Na seção 3.1 é apresentado o modelo de sistema adotado para esta rede de comunicação descrevendo como a arquitetura e a topologia da rede foi projetada.

Na seção 3.2 é apresentada a especificação formal em SDL dos nós da rede FlexRay. Esta seção é dividida em duas subseções que descrevem os componentes da arquitetura do nó. Na subseção 3.2.1 é mostrado como os mecanismos do núcleo do protocolo foram integrados dentro do bloco do controlador de comunicação (***Communication_Controller***). A subseção 3.2.2 apresenta como o controlador de barramento (***Bus_Driver***) foi criado e especificado formalmente em SDL.

Para simular uma rede completa fez-se necessária a modelagem em SDL do meio físico dentro do sistema. Sendo assim o barramento foi implementado de forma simples e eficiente no bloco ***Bus***. A seção 3.3 apresenta a descrição do comportamento interno do bloco ***Bus*** e explicita seu funcionamento básico.

Na seção 3.4 são apresentadas algumas adaptações na especificação do FlexRay [5] a fim de possibilitar a elaboração desta rede de comunicação usando o protocolo FlexRay.

3.1 Modelo do Sistema

Há várias maneiras de se projetar uma rede FlexRay. Ela pode ser configurada como uma rede do tipo barramento com um ou dois canais, uma rede estrela com um ou dois canais ou várias outras combinações híbridas de topologia de barramento e estrela [5].

Uma rede FlexRay consiste de, no máximo, dois canais. Cada nó na rede pode ser conectado a um ou a ambos os canais. Em uma condição livre de falhas, todos os nós conectados a um mesmo canal são capazes de se comunicar entre si. O número de nós conectados a uma rede pode variar de dois a vinte e dois nós, dependendo de condições do hardware (características do cabo, conceito de terminação e taxa de transmissão) [18].

Neste trabalho é considerada uma rede de um único canal configurado na topologia de barramento interconectando três nós. A topologia de barramento foi escolhida por ser a configuração mais simples dentre todas. Já o número de nós foi dimensionado tendo por base o número mínimo de nós necessários para simular uma inicialização completa de uma rede FlexRay. O ato de começar o processo de inicialização da rede é chamado de *coldstart*. Somente alguns nós estão autorizados a inicializar a rede, chamados nós *coldstart*. De acordo com a especificação, três diferentes tipos de caminhos podem ser seguidos pelo nó dependendo de sua configuração (mais detalhes encontram-se na seção 4.2 do capítulo 4).

A partir da arquitetura interna de um nó FlexRay, descrita no capítulo 2, foi projetado o sistema FlexRay deste trabalho. Foi considerado neste trabalho um sistema com três nós interconectados através da topologia de barramento. O esboço da arquitetura de cada nó e o sistema de barramento FlexRay modelado estão ilustrados na Figura 3.1.

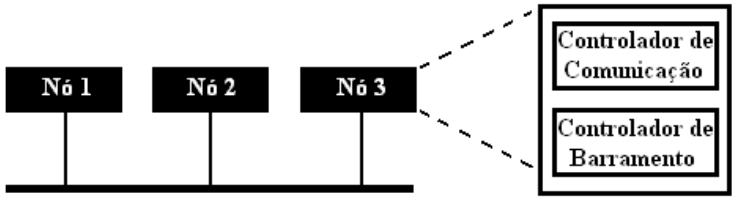


Figura 3.1: Esboço do sistema FlexRay modelado

Como ilustrado na Figura 3.1 cada nó constitui-se apenas dos controladores de comunicação e de barramento. Diferentemente da arquitetura do nó encontrada na especificação do protocolo a arquitetura aqui proposta não traz o host como componente do sistema. Dessa forma o sistema em questão transforma-se em um sistema FlexRay geral e pode ser usado para

diversas aplicações diferentes, já que o host – responsável pela especificidade da aplicação – não é modelado. Apesar do host não ser modelado como componente do sistema FlexRay sua interação com o sistema é feita de forma integral através do ambiente.

3.2 Especificação em SDL dos Nós da Rede FlexRay

O sistema modelado em SDL segue o mesmo padrão do esboço da Figura 3.2. A linguagem SDL serve tanto para representar o comportamento do sistema como sua estrutura, usando diferentes níveis de abstração desde o nível mais alto até o nível mais detalhado do sistema especificado.

Um sistema em SDL é dividido em blocos, que se comunicam uns com os outros e com o ambiente através de canais. Cada bloco é composto por um conjunto de processos que também se comunicam entre si através de rotas de sinal. O processo, por sua vez, trata-se do nível de abstração onde é especificado o comportamento do sistema através da definição de estados e transições.

O ambiente (*environment)* é o meio pelo qual um sistema SDL se comunica com o usuário. Os sinais trocados entre o sistema e o ambiente são responsáveis pela entrada e saída de dados do sistema. Sempre que necessário, podem ser especificados procedimentos (dentro de processos) que realizam alguma tarefa específica.

A Figura 3.2 ilustra o nível de sistema da rede FlexRay modelada. O sistema é uma entidade em SDL representada por um quadro que delimita a rede modelada. O que está fora do escopo deste quadro é chamado de ambiente e não está modelado em SDL. Os blocos na Figura 3.2 descrevem um conjunto de nós interconectados por um barramento. Os blocos chamados ***Node1***, ***Node2*** e ***Node3*** representam os nós na rede. O bloco ***Bus*** representa o meio físico, ou seja, o barramento.

Cada bloco que representa um nó é conectado a cinco canais. Esses canais permitirão ao sistema modelado interagir com outras partes não especificadas neste modelo, que são: o CHI e alguns dos mecanismos do núcleo do protocolo FlexRay (codificação/decodificação e sincronismo do clock).

O termo CHI será visto com muita freqüência ao longo desta dissertação e em algumas figuras do sistema. Este termo é muito usado em [5] e também foi adotado neste trabalho para designar a interface entre o *host* e o controlador de comunicação (*Controller Host Interface*).

Para aproximar o sistema FlexRay de um sistema real o meio físico (barramento) também foi modelado em SDL (Figura 3.2). O objetivo fundamental do bloco ***Bus*** (barramento) é retransmitir os sinais (*frame* ou símbolo) enviados pelos nós. Na seção 3.3 será mostrado detalhadamente como o bloco ***Bus*** foi modelado.

O processo de codificação e decodificação é representado pelo termo CODEC. O CODEC não teve seu processo integralmente modelado neste sistema por se compor de uma abstração de mais baixo nível. Dessa forma, o processo CODEC foi considerado fora do escopo deste sistema, mas sua interação com o sistema manteve-se através da comunicação pelo ambiente. O mesmo ocorreu com os processos de sincronismo do *clock*. O mecanismo de sincronismo do *clock* é dividido em três processos: o processo de inicialização do sincronismo do *clock* (CSS), o processo de sincronismo do *clock* (CSP) e o processo de geração de *macrotick* (MTG).

Todos esses três processos também interagem com o sistema através do ambiente e utilizam os canais para se comunicarem. Os sinais que trafegam em cada direção através dos canais são indicados por listas de sinais anexadas às extremidades dos canais. Os sinais que saem do sistema e entram nos processos não modelados são descritos nas listas de sinais como nome-do-processo_In (por exemplo, ***Csp_In***). Já os sinais que saem dos processos não modelados e entram no sistema são descritos nas listas de sinais como nome-do-processo_Out (por exemplo, ***Csp_Out***).

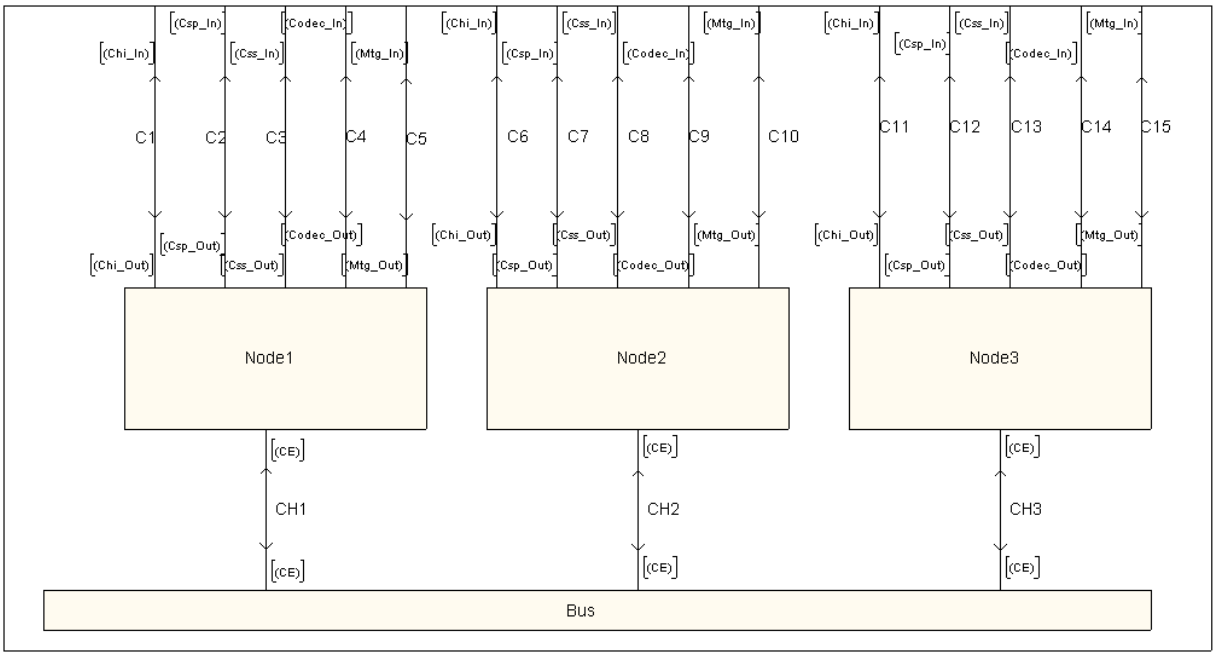


Figura 3.2: Sistema SDL representando a rede FlexRay

A comunicação entre os nós da rede é alcançada codificando a carga útil de um pacote de dados (enviada à priori pelo CHI) em frames ou símbolos e transmitindo através do barramento. Esses sinais referentes aos *frames* ou símbolos estão contidos na lista de sinal ***CE***. A sigla ***CE*** designa *Communication Element* (elemento de comunicação) e foi especificada desta forma em [5].

As descrições do comportamento dentro do nó 1 e do nó 2 são as mesmas. Somente o nó 3 tem descrição de comportamento diferente, pois ele foi adaptado para não ser um nó *coldstart*.

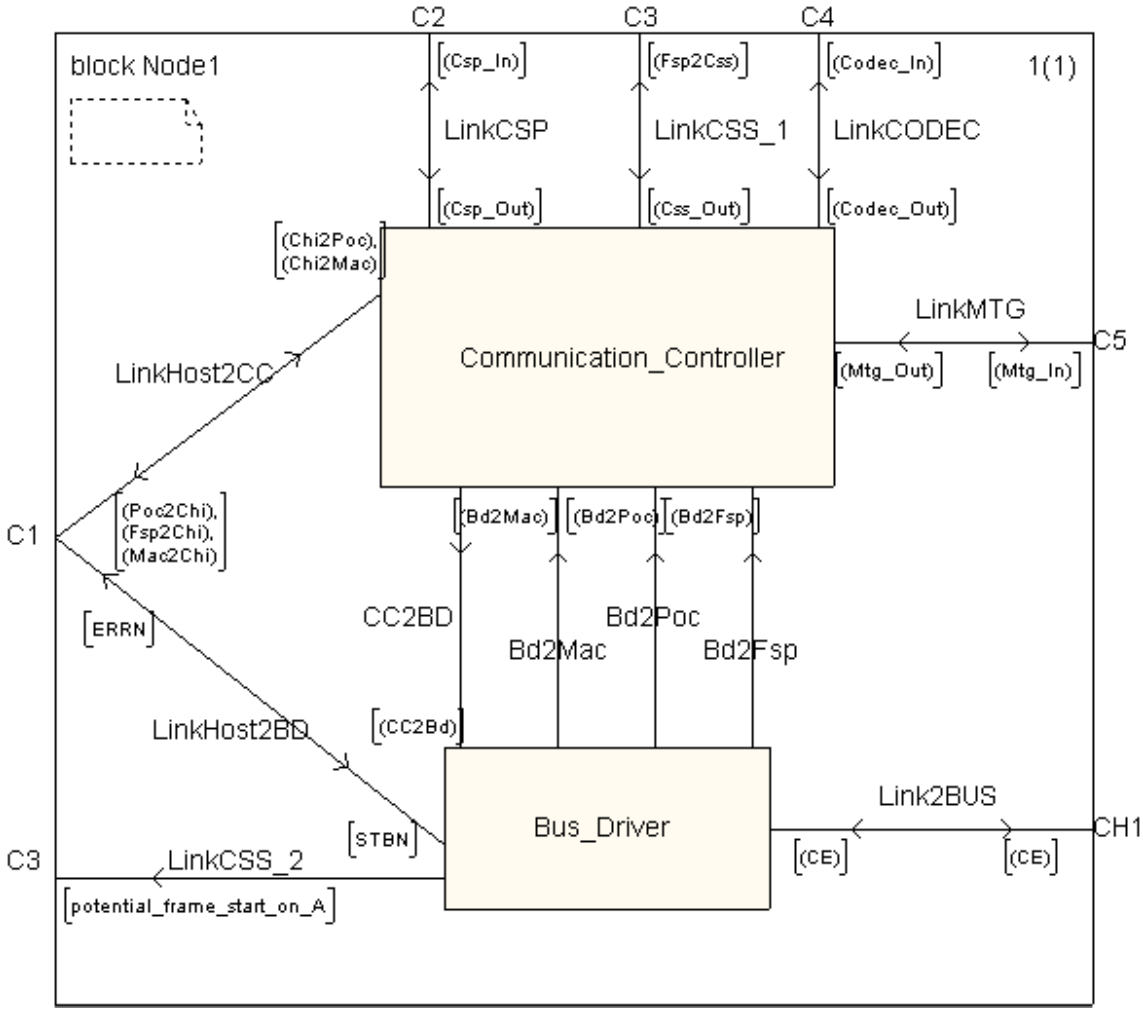


Figura 3.3: Especificação de um nó na rede FlexRay

A Figura 3.3 ilustra como um nó foi especificado em SDL. Cada bloco nó (***Node1***, ***Node2*** e ***Node3***) no sistema modelado contém um bloco controlador de comunicação (***Communication_Controller***) e um bloco controlador de barramento (***Bus_Driver***).

O bloco ***Communication_Controller*** comunica-se indiretamente e através de rotas de sinal (*signal routes*) com o CHI (***LinkHost2CC***) e com os processos CSP (***LinkCSP***), CSS (***LinkCSS_1***), CODEC (***LinkCODEC***) e MTG (***LinkMTG***).

Foi usado o termo *Link* para nomear as rotas de sinal que realizam essa comunicação indireta com os processos externos, por exemplo, ***LinkCSP***.

O bloco ***Bus_Driver*** troca sinais diretamente com o bloco ***Communication_Controller*** e indiretamente com o CHI (***LinkHost2BD***), com o processo CSS (***LinkCSS_2***) e com o barramento através do ambiente (***Link2BUS***).

Foram utilizados nomes diferentes para diferenciar as rotas de sinal direcionadas ao processo CSS, mas que partem de blocos distintos.

3.2.1 Controlador de Comunicação (Communication Controller)

A Figura 3.4 ilustra o bloco do controlador de comunicação (***Communication_Controller)***. Esse bloco recebe e envia sinais para o CHI, para o bloco do controlador de barramento (***Bus_Driver)*** e para os outros componentes do mecanismo do protocolo FlexRay (não-modelados). Três dos principais mecanismos que compõem o protocolo FlexRay estão descritos como processos dentro do bloco ***Communication_Controller***. Na Figura 3.4 também estão ilustrados os três processos do bloco ***Communication_Controller***: POC (Protocol Operation Control), MAC (Media Access Control) e FSP (Frame and Symbol Processing).

O processo POC será descrito na subseção 3.2.1.1. O processo POC está conectado a nove rotas de sinal. Sete delas se conectam ao bloco ***Communication_Controller*** e as outras duas se conectam aos processos MAC e FSP. O POC está conectado a todos os processos do núcleo do protocolo. As setas indicam o sentido da transmissão nas rotas de sinal, assim como nos canais.

O processo MAC, descrito na subseção 3.2.1.2, está conectado a oito rotas de sinal. Uma delas, R5, faz conexão do host com o MAC e permite que a aplicação corrente envie os dados que serão codificados e transmitidos pelo barramento. As rotas R1 e R21 conectam o MAC ao controlador de barramento, a rota R13 conecta o MAC ao FSP, a rota R12 conecta o MAC ao processo não-modelado CODEC, a rota R3 conecta o MAC ao POC, a rota R18 conecta o MAC ao processo não-modelado CSP e a rota R9 conecta o MAC ao processo não-modelado MTG.

Todas as rotas de sinal carregam um ou mais sinais de/para o processo MAC.

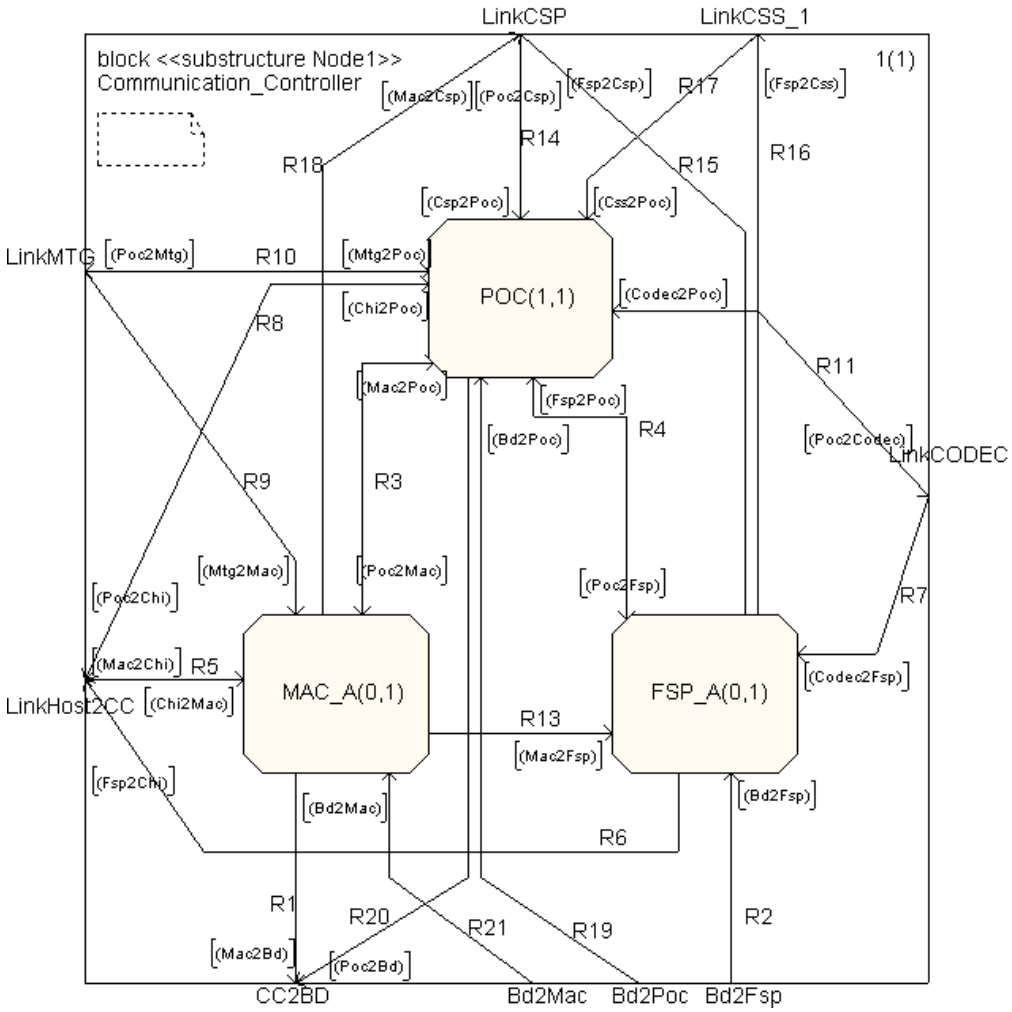


Figura 3.4: Especificação do bloco Controlador de Comunicação

O processo FSP (descrito na subseção 3.2.1.3) está conectado a sete rotas de sinal. A rota R6 faz conexão do FSP ao *host* e permite que os dados recebidos por *frames* ou símbolos decodificados sejam enviados para a aplicação corrente. A rota R2 conecta o controlador de barramento ao FSP, a rota R13 conecta o MAC ao FSP, a rota R7 conecta o processo não-modelado CODEC ao FSP, a rota R4 conecta o FSP ao POC, a rota R16 conecta o FSP ao processo não-modelado CSS e a rota R15 conecta o FSP ao processo não-modelado CSP. Todas as rotas de sinal carregam um ou mais sinais de/para o processo FSP.

3.2.1.1 POC (Protocol Operation Control)

O POC é o principal processo no mecanismo de funcionamento do protocolo FlexRay. Sua finalidade é reagir aos comandos do host e às condições do protocolo ativando mudanças

coerentes nos mecanismos de maneira síncrona, e fornecer ao *host* o estado atual dos mecanismos em relação às mudanças. Portanto, é ele quem controla todas as ações dos demais processos do núcleo do protocolo.

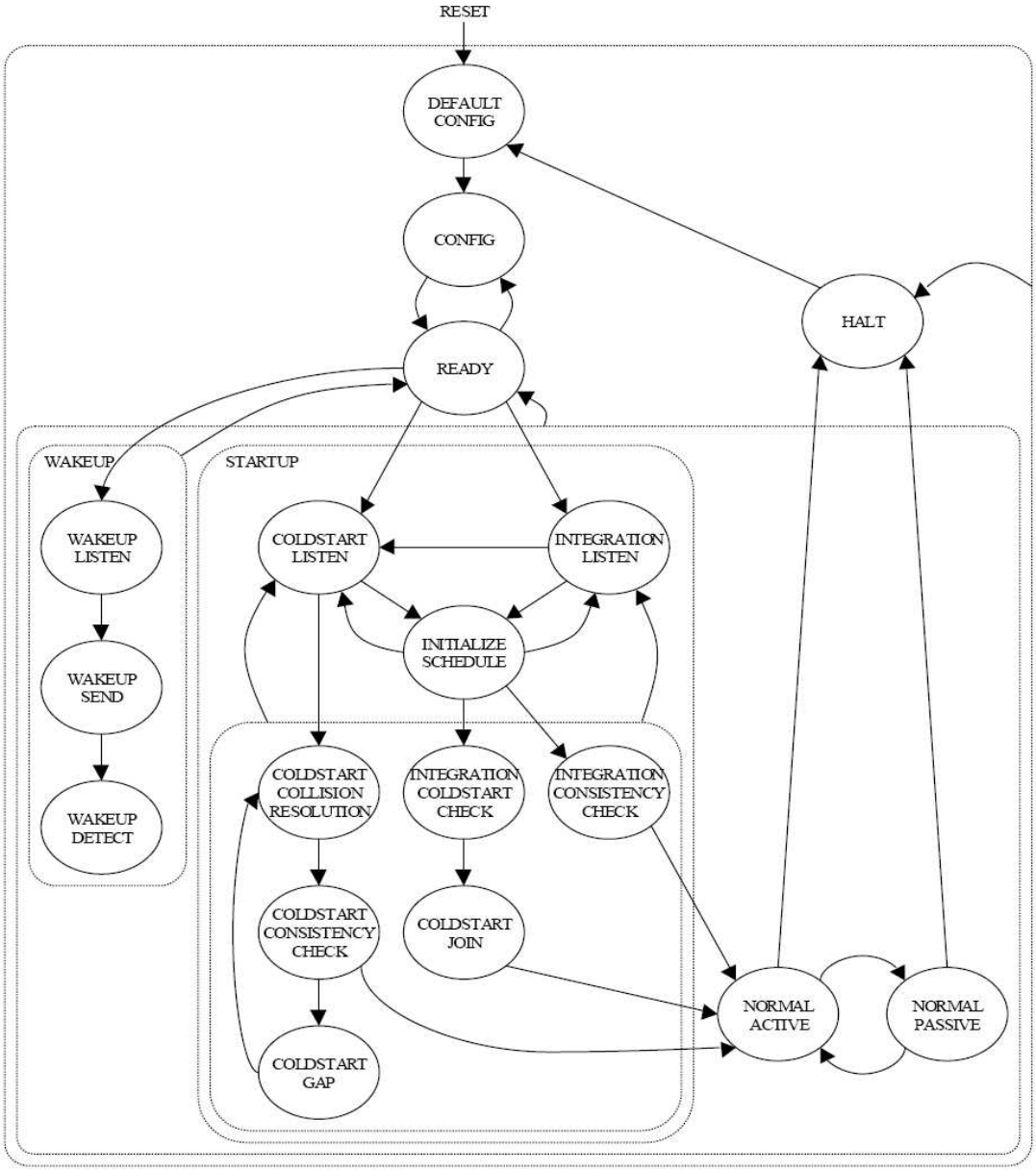


Figura 3.5: Visão geral das transições de estado do processo POC [24]

A Figura 3.5 mostra a seqüência de etapas necessárias para uma rede FlexRay passar do modo inicial (estado *DEFAULT_CONFIG*) para o modo de operação normal (estado *NORMAL_ACTIVE*) através do diagrama de transição de estado do processo POC. O POC contém 18 estados no total, sendo que 12 deles encontram-se embutidos em dois procedimentos:

Wakeup e *Startup*. O procedimento *wakeup* é executado após o nó ser configurado através da passagem pelos estados *DEFAULT_CONFIG*, *CONFIG* e *READY*. Esse procedimento é responsável por ativar os outros nós interligados à rede. Após sair do procedimento *wakeup* o POC retorna ao estado *READY* e daí entra no procedimento *startup*. Esse procedimento engloba o mecanismo de inicialização da rede e, assim, o POC passa por vários estados até estar sincronizado com os outros nós e então alcançar o estado *NORMAL_ACTIVE*.

O processo POC é responsável pelo mecanismo de inicialização do sistema e também pela coordenação de todo o comportamento do protocolo através do interfaceamento entre o *host* e os mecanismos base de codificação/decodificação, controle de acesso ao meio, processamento de *frame*/símbolo e sincronismo do *clock*.

A descrição em SDL do comportamento interno dos processos POC, MAC e FSP baseou-se em [5], porém algumas alterações e contribuições foram inseridas. Tais alterações estão detalhadas na seção 3.4 e nas subseções de cada processo (subseções 3.2.1.1, 3.2.1.2 e 3.2.1.3).

Uma alteração importante no processo POC refere-se ao procedimento *Startup*.

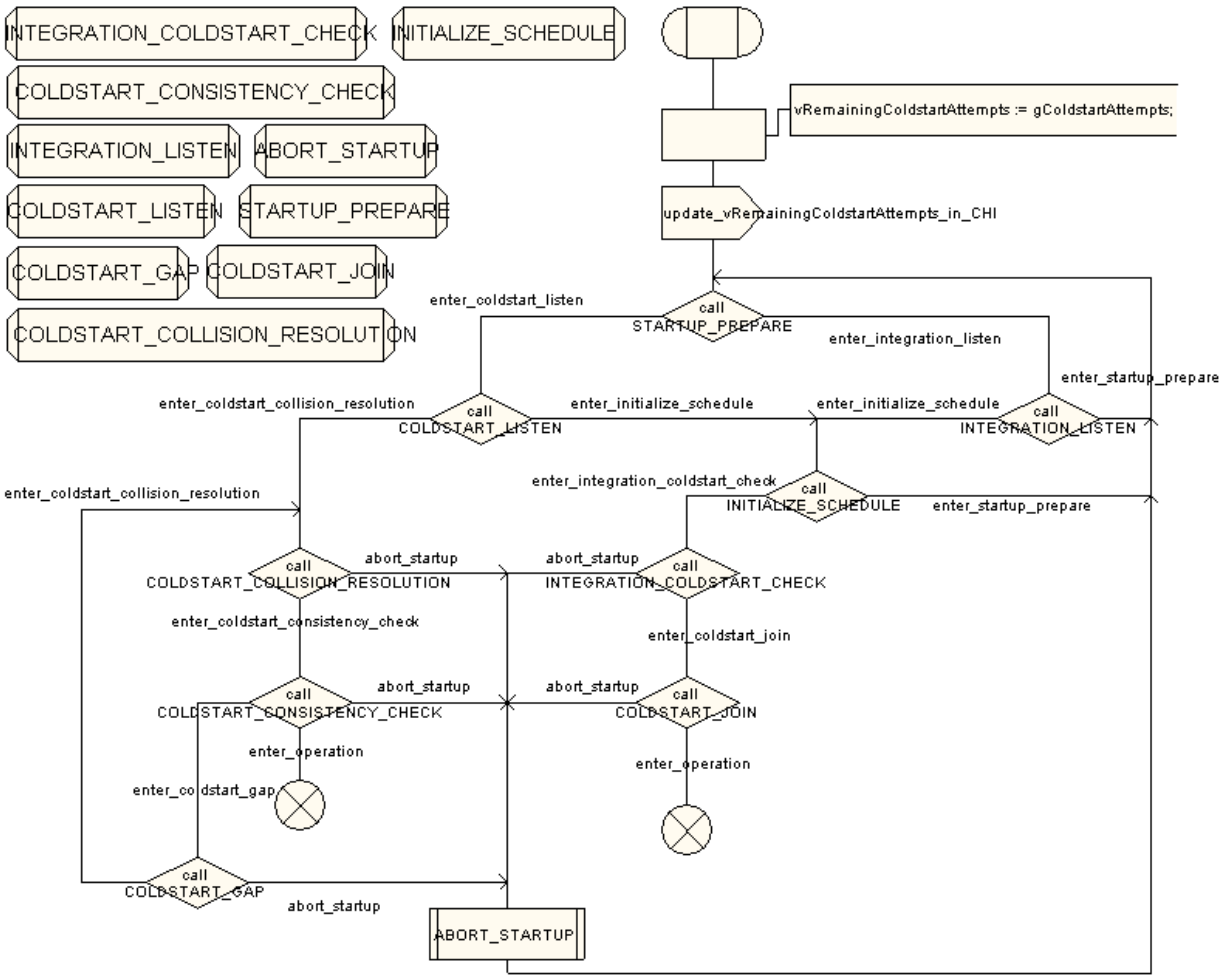


Figura 3.6: Especificação do procedimento Startup para os nós 1 e 2

A Figura 3.6 ilustra como o procedimento *Startup* foi especificado para os nós 1 e 2. A especificação em SDL dos nós 1 e 2 são similares. Somente o nó 3 teve uma especificação diferente, pois ele não foi adaptado para ser um nó *coldstart*. A principal mudança na especificação do nó 3 está localizada no procedimento *Startup*, como ilustrada na Figura 3.7. Como o nó 3 não pode inicializar a rede, todos os procedimentos relacionados a esta função foram eliminados a fim de otimizar o funcionamento do sistema. Dessa forma, o procedimento *Startup* do nó 3 só possui metade dos procedimentos encontrados nos nós 1 e 2. Porém, existe um procedimento exclusivo do nó 3 que é o *INTEGRATION_CONSISTENCY_CHECK*. Somente nós que não são *coldstart* passam por este procedimento. Nesse procedimento o nó compara se seu *clock* está sincronizado com o nó *coldstart* (nó que inicializou a rede) e verifica se existem, pelo menos, dois nós enviando *frames startup* (*frames* de inicialização e sincronismo). Caso os

requisitos do procedimento sejam cumpridos o procedimento *Startup* é finalizado e o nó entra em modo de operação.

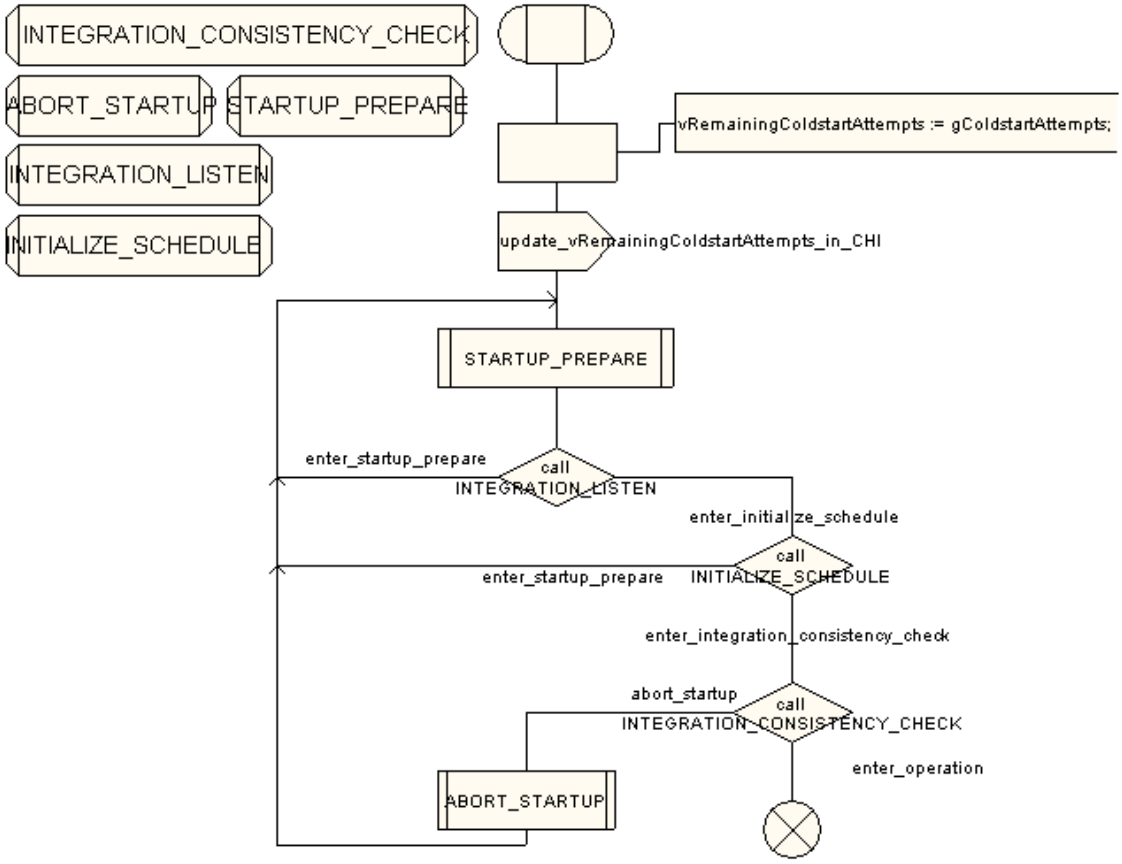


Figura 3.7: Especificação do procedimento Startup para o nó 3

3.2.1.2 MAC (Media Access Control)

O MAC define o controle de acesso ao meio físico (barramento) entre os nós conectados ao canal de comunicação. Além de gerenciar o compartilhamento do acesso ao barramento o MAC também gerencia o ciclo de comunicação recorrente. O ciclo de comunicação é o elemento fundamental do esquema de acesso ao meio dentro do FlexRay.

A especificação em SDL do comportamento interno do processo MAC baseou-se em [5]. Porém, foram feitas duas alterações importantes na modelagem em SDL do processo MAC. A primeira alteração refere-se à forma como a mudança de modo de operação era processada no processo MAC e a segunda alteração refere-se à especificação do processo MAC para o nó 3.

A Figura 3.8 mostra como a implementação do processo MAC foi sugerida em [5]. Essa figura traz à esquerda o diagrama com o comportamento principal do processo MAC e à direita o

diagrama com o mecanismo responsável por capturar o comando de mudança do modo de operação no processo MAC.

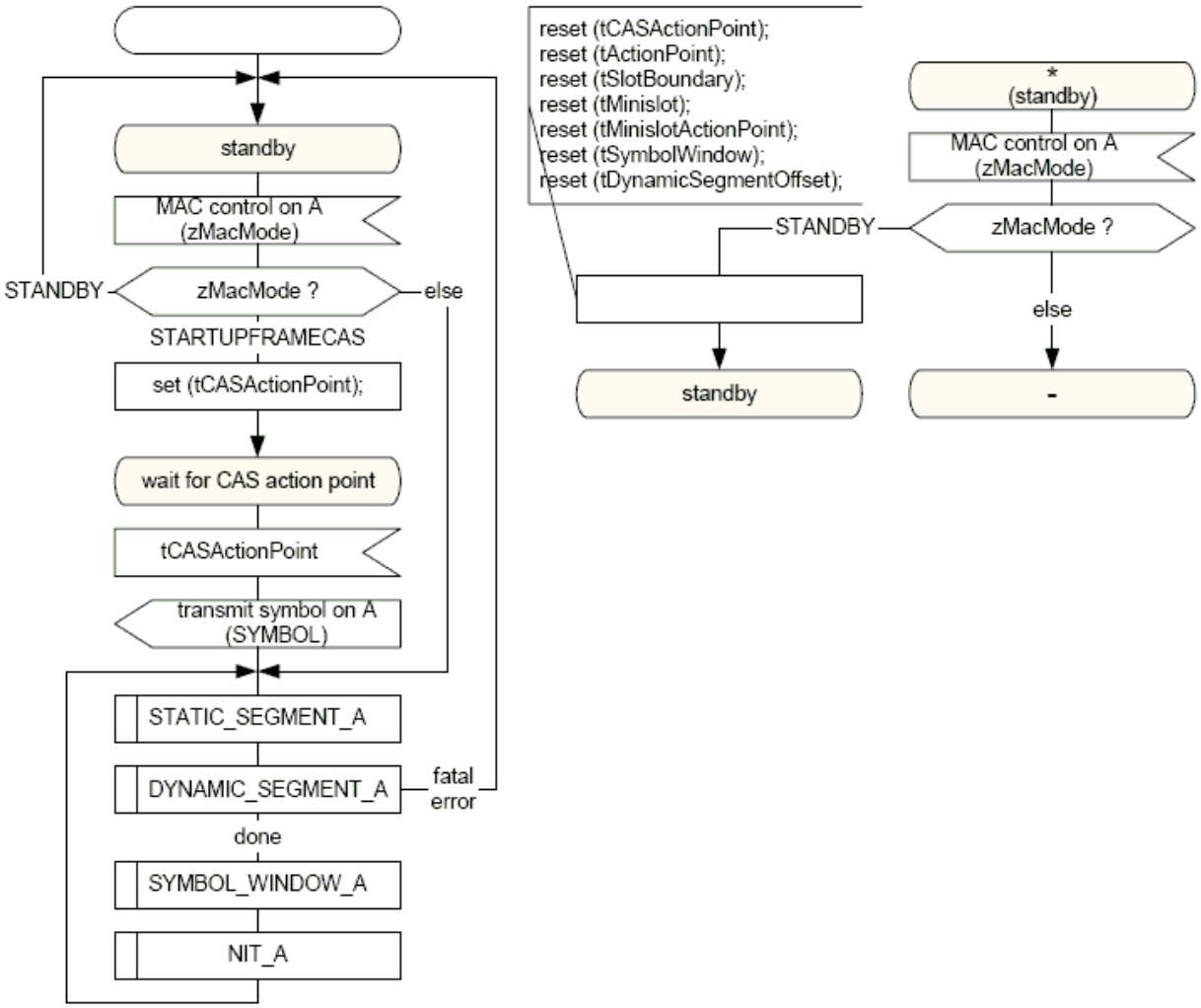


Figura 3.8: Diagrama do processo MAC descrito em [5]

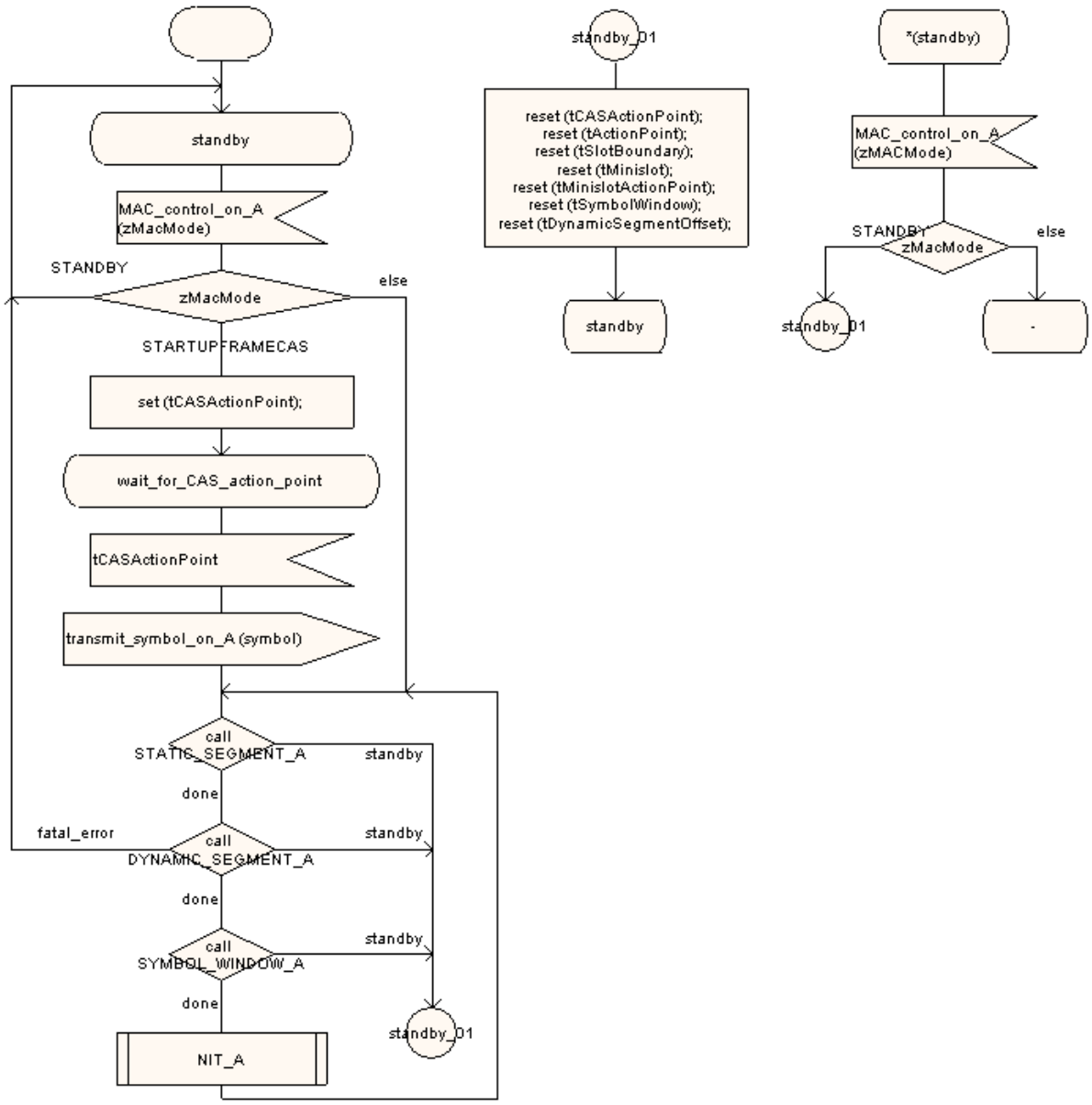


Figura 3.9: Especificação do processo MAC para os nós 1 e 2

A Figura 3.9 mostra como realmente o processo MAC foi especificado neste trabalho para se tornar funcional. O objetivo em mostrar as Figuras 3.8 e 3.9 é comparar o mecanismo de mudança do modo de operação do processo. Pela definição em [5] cada processo do controlador de comunicação deve ter um artifício de processamento da mudança de modo de operação, indiferentemente do estado em que o processo se encontre. O problema é que o processo MAC foi especificado fazendo grande uso de construções de procedimentos (*procedure*). E na linguagem SDL, quando um procedimento é chamado, o processo que o chamou fica bloqueado

aguardando o final da execução do procedimento. Dessa forma se um comando de modo de operação chegar ao processo e um procedimento estiver em execução o comando será descartado, já que o processamento da mudança do modo de operação era realizado apenas no processo e não dentro de cada procedimento. Uma solução encontrada foi adicionar em cada procedimento uma entrada especial para o comando de mudança de modo de operação, como visto na Figura 3.10 para o procedimento ***SYMBOL_WINDOW_A***. Assim, se algum procedimento em execução receber tal comando ele o processará corretamente.

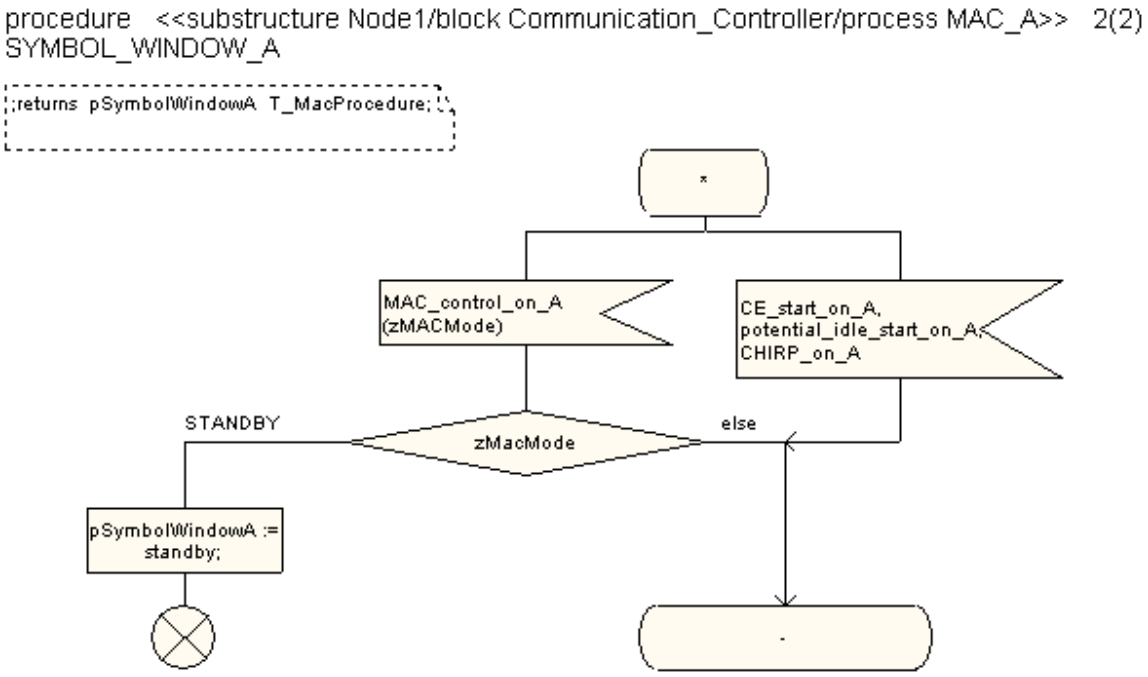


Figura 3.10: Comando de modo de operação dos procedimentos do MAC

O sinal ***MAC_control_on_A*** da Figura 3.10 recebe o parâmetro ***zMacMode*** que é responsável por carregar o modo de operação a ser alterado no processo. Caso a mudança seja para o modo *STANDBY* o procedimento deverá cancelar sua execução e retornar ao processo (retornando o valor *standby*) para executar as operações necessárias. Há uma alternativa com o valor *standby* na saída dos segmentos do ciclo de comunicação que os direcionam para as tarefas corretas e corrige o problema (Figura 3.9).

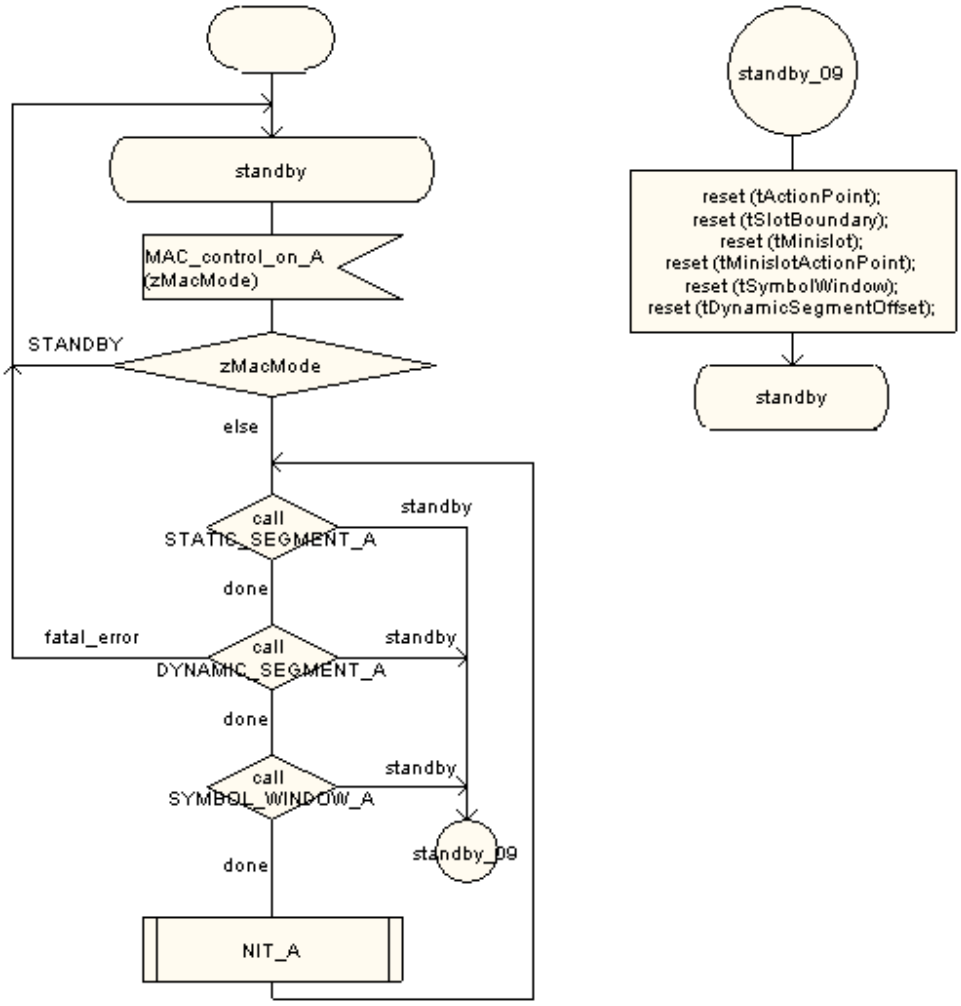


Figura 3.11: Especificação do processo MAC para o nó 3

A Figura 3.11 traz a segunda alteração feita no processo MAC. Essa alteração refere-se à especificação do nó 3 e está diretamente relacionada ao fato do mesmo não ser um nó *coldstart*. Ao ser inicializado o processo MAC entra no estado ***standby*** e recebe um sinal do POC configurando-o para um determinado modo de operação (Figura 3.9). A partir de então o MAC verifica em qual modo de operação ele se encontra. Se o modo for ***STANDBY***, ele retorna ao estado ***standby***; se o modo for ***STARTUPFRAMECAS***, ele realiza algumas tarefas relativas à inicialização da rede e depois inicia o ciclo de comunicação começando pelo procedimento ***STATIC_SEGMENT_A*** (segmento estático); se o modo for diferente de ***STANDBY*** e ***STARTUPFRAMECAS*** o MAC inicia o ciclo de comunicação. Na especificação da Figura 3.11 não há uma alternativa para o modo ***STARTUPFRAMECAS*** na caixa de decisão ***zMacMode***, como acontece na Figura 3.9 (processo MAC para nó 1 e 2) . Como o nó 3 não pode inicializar

uma rede este modo de operação nunca aconteceria, e, desta forma, esta condição nunca seria alcançada. Logo, a parte que se refere à inicialização da rede foi removida do diagrama do processo MAC do nó 3.

3.2.1.3 FSP (Frame and Symbol Processing)

O FSP é o principal processo de decodificação de *frame* e símbolo no protocolo FlexRay. O FSP verifica a correta temporização dos *frames* e símbolos em relação ao esquema TDMA, aplica testes sintáticos adicionais nos *frames* recebidos e verifica a corretude semântica dos *frames* recebidos.

A especificação do processo FSP, assim como aconteceu na especificação dos processos POC e MAC, não diferiu da especificação descrita em [5]. Existiu, porém, uma exceção. Trata-se de uma entrada de sinal proveniente do canal de comunicação B. Conforme visto no início deste capítulo, foi considerada neste trabalho a utilização de apenas um canal de comunicação. Por conseguinte, qualquer implementação referente a um segundo canal de comunicação foi desconsiderado.

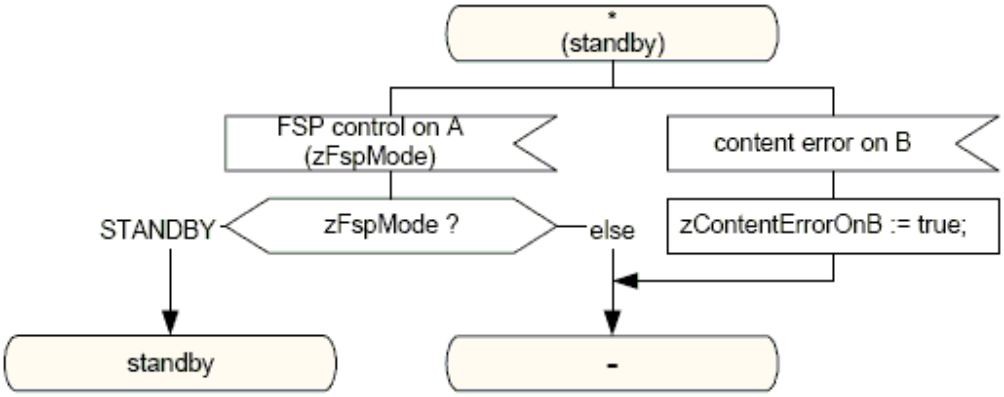


Figura 3.12: Exemplo de um diagrama referente a um segundo canal de comunicação [5]

A Figura 3.12 ilustra um diagrama do processo FSP, encontrado em [5], que contém a descrição de uma rede com duplo canal de comunicação. Nessa figura há uma entrada de um sinal ***content error on B***. Este sinal mencionado refere-se a um erro de conteúdo específico do canal B, ou seja, um segundo canal de comunicação. Como tal implementação somente seria necessária se a rede modelada utilizasse dois canais de comunicação, este comportamento foi removido. A Figura 3.13 ilustra como o mesmo diagrama da Figura 3.12 foi especificado na rede FlexRay deste trabalho.

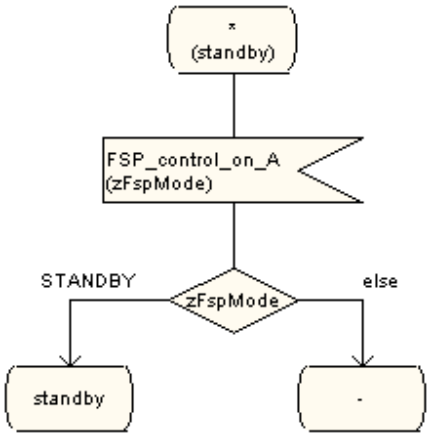


Figura 3.13: Especificação do diagrama da Figura 3.12

Outras alterações relevantes (feitas nos processos FSP, MAC e POC) serão detalhadas na seção 3.4, visto que tais alterações se repetem na especificação de toda a rede.

3.2.2 Controlador de Barramento (Bus Driver)

O controlador de barramento é um componente eletrônico que consiste de um transmissor e um receptor que conecta o controlador de comunicação a um canal de comunicação.

O controlador de barramento (*Bus Driver*) não é descrito em [5] usando a linguagem SDL, assim como acontece para o controlador de comunicação. O consórcio FlexRay [1] disponibiliza apenas alguns detalhes do controlador de barramento e sua ligação com o *host* e com o controlador de comunicação. Há também outra especificação do sistema FlexRay específico para a camada física [19], que detalha os componentes do hardware do sistema FlexRay. No entanto, por ser um componente importante da arquitetura da rede FlexRay, o controlador de barramento foi especificado neste trabalho usando a linguagem SDL.

O objetivo da especificação deste controlador é simular o funcionamento do mesmo para assim poder criar uma completa e funcional rede FlexRay. Informações significativas para a implementação do bloco ***Bus_Driver*** foram retiradas de [5], tais como: sinais trocados entre o *host* e o controlador de barramento, modos de operação do controlador de barramento e a importante função, atribuída ao controlador de barramento, de ativar os componentes do controlador de comunicação assim que recebe um sinal de *wake-up* enviado por outro nó da rede.

A partir das informações coletadas o bloco ***Bus_Driver*** foi especificado contendo dois processos: o processo transmissor (***Transmitter***) e o processo receptor (***Receiver***). Para simplificar o sistema FlexRay, mas ainda seguindo as regras de operação do protocolo, o controlador de barramento também foi projetado para realizar parte do trabalho do processo CODEC. O CODEC é um processo responsável por codificar e decodificar todo *frame* ou símbolo que o nó transmite ou recebe. Devido a essa importante função, poderia ser questionada a não implementação deste processo. A explicação para isso reside na complexidade do comportamento deste processo. Para implementá-lo de forma satisfatória seria necessário implementar todos os processos que não foram considerados neste trabalho, visto que os mesmos são interdependentes e lidam com mecanismos de sincronização. A especificação de todos os processos que fazem parte do protocolo FlexRay seria um trabalho muito complexo e não agregaria nenhum valor substancial se implementado neste trabalho. Como o objetivo do projeto era modelar uma rede funcional foi idealizado um esquema de adaptação do comportamento do

processo CODEC para embutí-lo no controlador de barramento. Dessa forma eliminou-se a complexidade do processo sem perder sua utilidade.

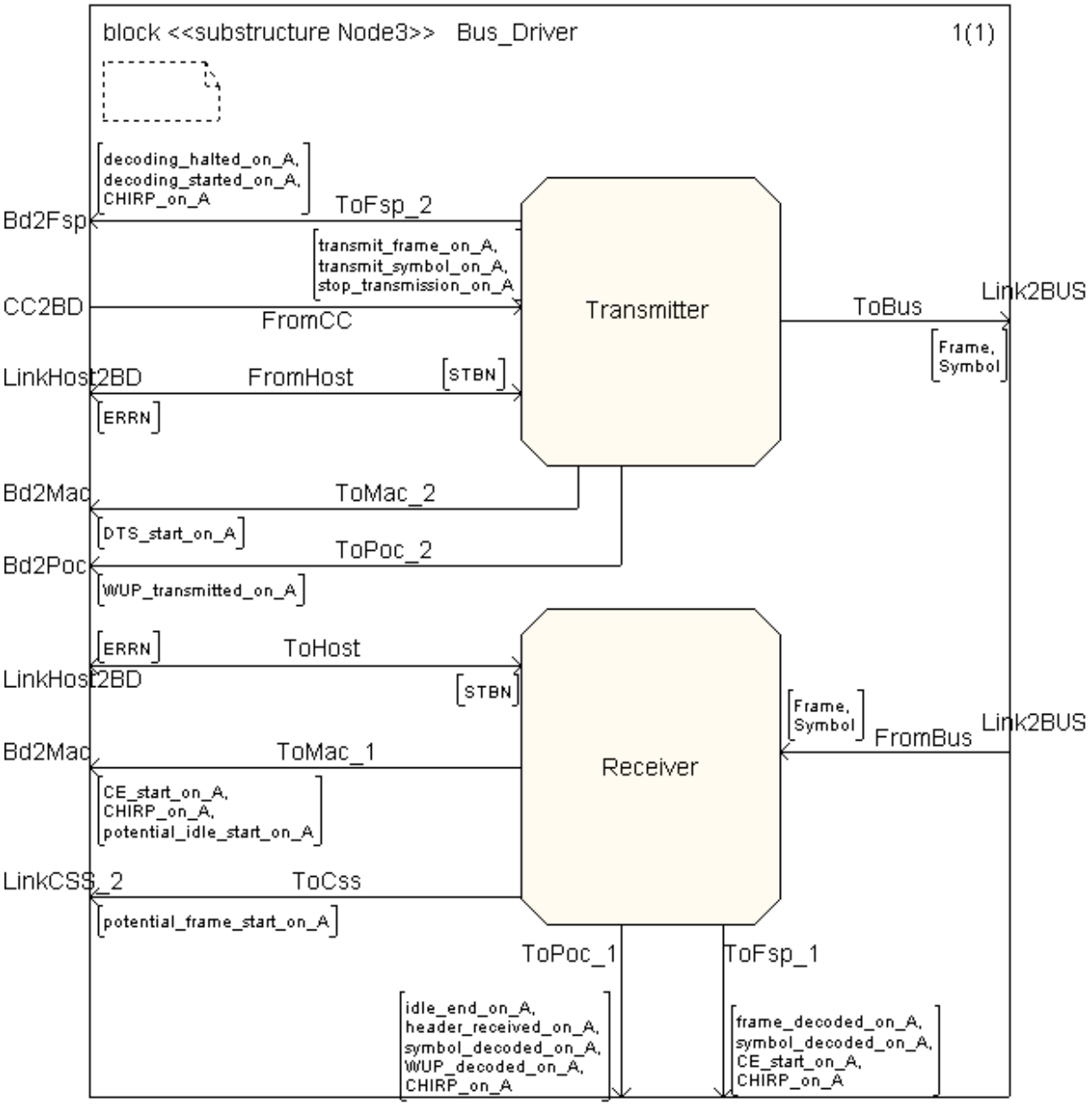


Figura 3.14: Especificação do bloco do controlador de barramento (***Bus_Driver***)

A Figura 3.14 mostra o diagrama do bloco ***Bus_Driver*** contendo seus processos ***Transmitter*** e ***Receiver*** e suas respectivas rotas de sinal. Em linhas gerais, o controlador de barramento funciona da seguinte maneira: quando o sinal ***transmit_frame_on_A (vType, vTF)*** chega ao bloco ***Bus_Driver***, é guiado através da rota de sinal ***FromCC*** para o processo

Transmitter e depois é convertido para o sinal ***Frame (vType, vTF)*** e finalmente enviado para o barramento. No processo ***Receiver*** ocorre a conversão inversa. Quando o sinal ***Frame (vType, vTF)*** chega, seu parâmetro ***vTF*** (variável que contém os dados do *frame* a ser transmitido) é atribuído ao parâmetro ***vRF*** (variável que contém os dados do *frame* recebido) e um sinal ***frame_decoded_on_A (vRF)*** é enviado. O parâmetro ***vType*** (variável que contém o tipo de transmissão) é usado para diferenciar a recepção do elemento de comunicação.

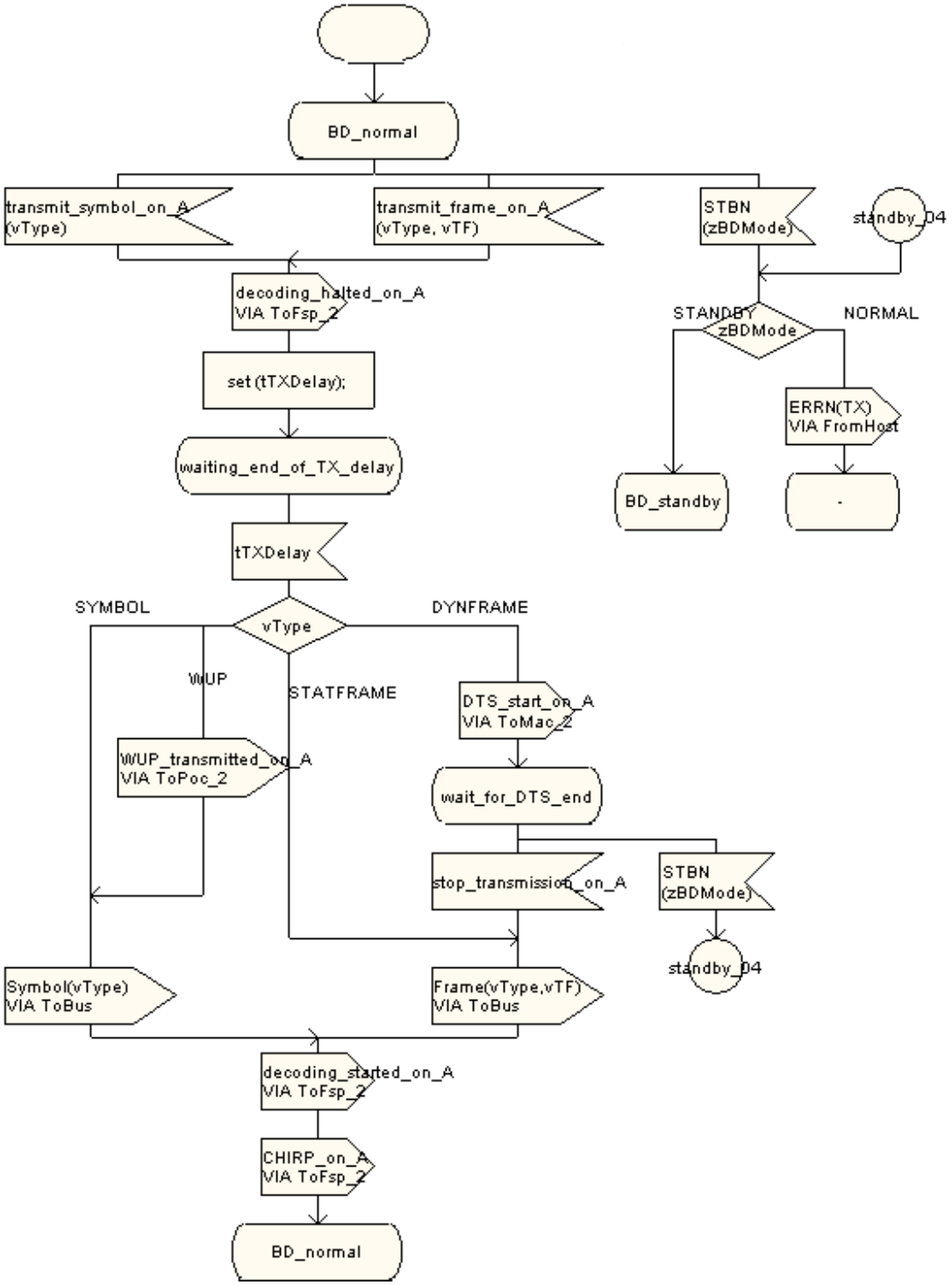


Figura 3.15: Especificação do processo transmissor do controlador de barramento

Na Figura 3.15 está ilustrado o processo transmissor do controlador de barramento. O processo ***Transmitter*** interage com o ambiente através de seis rotas de sinal. As rotas ***ToFsp_2,***

FromCC, ToMac_2 e ***ToPoc_2*** carregam sinais relacionados ao processo de codificação e decodificação do CODEC. Já as rotas ***FromHost*** e ***ToBus*** são exclusivas do controlador de barramento e transmitem, respectivamente, sinais de/para o *host* e sinais para o barramento.

O processo ***Receiver***, assim como o ***Transmitter***, também interage com o ambiente através de seis rotas de sinal, sendo: **ToPoc_1, ToMac_1, ToFsp_1** e **ToCss** as rotas relacionadas ao CODEC e ***ToHost*** e ***FromBus*** as rotas exclusivas do controlador de barramento.

Tanto o processo ***Transmitter*** quanto o processo ***Receiver*** são constituídos basicamente de dois estados: ***BD_normal*** e ***BD_standby***. Ambos os processos iniciam sua operação no estado ***BD_normal***, que é o estado de operação normal do controlador de barramento.

O processo ***Transmitter*** inicia sua operação aguardando no estado ***BD_normal*** por um sinal de entrada. Se o sinal recebido for o ***transmit_symbol_on_A(vType)*** ou o ***transmit_frame_on_A(vType)***, proveniente do processo MAC do controlador de comunicação, o processo enviará um sinal ***decoding_halted_on_A*** para o processo FSP através da rota de sinal ***ToFsp_2***. Em seguida o processo ***Transmitter*** passa por um mecanismo de atraso de transmissão. Este mecanismo foi criado para simular o tempo de codificação do processo CODEC. Posteriormente ocorre uma verificação do tipo de elemento de comunicação que será enviado. Dependendo do tipo encontrado um caminho diferente será seguido e, subseqüentemente, serão enviados ao processo FSP mais dois sinais: ***decoding_halted_on_A*** e ***CHIRP_on_A***.

No caso do sinal ***transmit_symbol_on_A(vType)***, que pode carregar apenas dois tipos de elementos de comunicação (*symbol* ou WUP), apenas dois caminhos poderão ser percorridos. Se na verificação for constatado o tipo ***SYMBOL*** (símbolo) o sinal a ser enviado pelo barramento (rota ***ToBus***) será o ***Symbol(vType)***, sendo que o parâmetro ***vType*** conterá o valor ***SYMBOL***. Se na verificação for constatado o elemento do tipo ***WUP*** o sinal ***WUP_transmitted_on_A*** será enviado para o processo POC e, após isso, o mesmo procedimento adotado para o tipo ***SYMBOL*** será seguido, apenas diferindo no parâmetro ***vType*** que conterá o valor ***WUP.***

No caso do sinal ***transmit_frame_on_A(vType, vTF)*** dois tipos distintos de elementos de comunicação podem ser encontrados (***STATFRAME*** e ***DYNFRAME***) e, dessa forma, apenas dois caminhos poderão ser percorridos. Se na verificação for constatado o tipo ***STATFRAME*** (*frame* do segmento estático) o sinal a ser enviado pelo barramento (rota ***ToBus***) será o ***Frame(vType, vTF)***, sendo que o parâmetro ***vType*** conterá o valor ***STATFRAME*** e o parâmetro ***vTF*** conterá o conteúdo do frame. Já se for verificado a presença do elemento ***DYNFRAME***

(*frame* do segmento dinâmico) o sinal ***DTS_start_on_A*** será enviado para o processo MAC através da rota ***ToMac_2*** e o processo ***Transmitter*** entrará no estado ***wait_for_DTS_end***. Nesse estado são aguardados dois sinais; se o sinal recebido for o ***stop_transmission_on_A***, proveniente do processo MAC, será enviado pelo barramento (rota ***ToBus***) o sinal ***Frame(vType, vTF)***, sendo que o parâmetro ***vType*** conterá o valor ***DYNFRAME*** e o parâmetro ***vTF*** conterá o conteúdo do *frame*; se o sinal recebido for o ***STBN(zBDMode)***, proveniente do *host*, o processo passará por uma verificação do modo de operação do controlador de barramento. Se o modo de operação for ***STANDBY*** o processo ***Transmitter*** passará para o estado ***BD_standby***. Caso o modo encontrado no parâmetro seja o ***NORMAL***, então o sinal ***ERRN(TX)*** será enviado ao host via rota ***FromHost*** e o processo continuará no estado ***BD_normal***.

Os sinais ***ERRN*** e ***STBN*** são responsáveis pela comunicação existente entre o controlador de barramento e o *host*. A especificação padrão da camada física do protocolo FlexRay [19] prevê a existência de uma interface entre o *host* e o controlador de barramento, porém os sinais dessa interface foram descritos em termos de sinais elétricos (níveis de tensão). Logo, para entrar em conformidade com a especificação em SDL, tanto o sinal ***STBN*** quanto o ***ERRN*** recebeu um parâmetro de entrada para identificar seu valor. O sinal ***STBN*** (*Standby Not*) permite ao *host* controlar os modos de operação do controlador do barramento e possui um parâmetro chamado ***zBDMode*** que carrega o valor do modo de operação, podendo ser ***NORMAL*** ou ***STANDBY***. Já o sinal ***ERRN*** (*Error Not*) é usado para indicar erro quando o controlador de barramento não está no modo de operação ordenado pelo *host*. O sinal ***ERRN*** possui um parâmetro para indicar se o erro aconteceu no processo transmissor (valor do parâmetro ***TX***) ou se o erro aconteceu no processo receptor (valor do parâmetro ***RX***).

Se o sinal recebido no estado inicial ***BD_normal*** tiver sido o ***STBN(zBDMode)***, então o processo passará por uma verificação do modo de operação do controlador de barramento seguindo o mesmo procedimento descrito anteriormente.

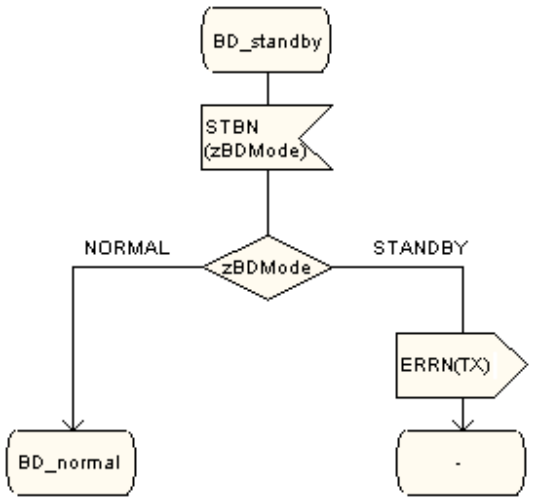


Figura 3.16: Estado ***BD_standby*** do processo Transmitter

A Figura 3.16 ilustra o diagrama que implementa o estado ***BD_standby*** do processo transmissor. Nesse estado o processo aguarda a chegada do sinal ***STBN(zBDMode)***, proveniente do host, e assim que esse sinal é consumido ocorre uma verificação do modo de operação contido na variável ***zBDMode***. Se o modo for o ***NORMAL***, então o processo passará para o estado ***BD_normal***; se o modo for o ***STANDBY***, então o sinal ***ERRN(TX)*** é enviado indicando erro do modo de operação do controlador de barramento. O processo ***Receiver*** possui um diagrama similar para descrever seu comportamento no estado ***BD_standby***, a única diferença está no parâmetro do sinal ***ERRN*** enviado, que terá o valor ***RX*** ao invés do ***TX***.

Na Figura 3.17 está ilustrado o processo receptor do controlador de barramento. O processo ***Receiver*** inicia sua operação aguardando no estado ***BD_normal*** por um sinal de entrada (*Frame* e *Symbol*) proveniente do barramento ou do *host* (STBN). Se o sinal recebido for o ***Frame(vType,vTF)*** ou o ***Symbol(vType)***, o processo enviará um sinal ***idle_end_on_A*** para o processo POC através da rota de sinal ***ToPoc_1*** e um sinal ***CE_start_on_A*** para os processos MAC e FSP através das rotas de sinal ***ToMac_1*** e ***ToFsp_1***, respectivamente. O processo então realiza uma verificação do tipo de elemento de comunicação recebido. Dependendo do tipo encontrado um caminho diferente será seguido e, subseqüentemente, serão enviados mais dois sinais: ***potential_idle_start_on_A*** para o processo MAC e ***CHIRP_on_A*** para os processos POC, MAC e FSP.

Se o sinal recebido for o ***Frame(vType,vTF)***, na verificação do ***vType*** o caminho a ser seguido será aquele relacionado aos termos ***STATFRAME*** e ***DYNFRAME***. Assim, o sinal

potential_frame_start_on_A será enviado para o processo CSS (ambiente) através da rota ***ToCss***. Após isso, ocorre a decodificação do *frame* recebido e então mais dois sinais são enviados: ***header_received_on_A*** (rota ***ToPoc_1***) e ***frame_decoded_on_A(vRF)*** (rota ***ToFsp_1***). Tais procedimentos foram herdados do processo CODEC e o conteúdo do frame decodificado é armazenado no parâmetro ***vRF*** para depois ser enviado pelo sinal correspondente.

Se o sinal recebido for o ***Symbol(vType)*** o processo passará por duas verificações do elemento de comunicação. Se na primeira verificação o elemento constatado for o ***WUP***, então o sinal ***WUP_decoded_on_A*** será enviado ao POC pela rota ***ToPoc_1*** e o processo retornará ao estado ***BD_normal***; caso o elemento não seja o ***WUP*** o processo seguirá pelo caminho devido até chegar à segunda verificação onde será constatado a existência do elemento ***SYMBOL***, e assim, o sinal ***symbol_decoded_on_A*** será enviado para os processos FSP (rota ***ToFsp_1***) e POC (rota ***ToPoc_1***).

Se o sinal recebido no estado inicial ***BD_normal*** tiver sido o ***STBN(zBDMode)***, então o processo passará por uma verificação do modo de operação seguindo o mesmo procedimento descrito no processo ***Transmitter***. Se o modo de operação for ***STANDBY*** o processo ***Receiver*** passará para o estado ***BD_standby***. Caso o modo encontrado no parâmetro seja o ***NORMAL***, então o sinal ***ERRN(RX)*** será enviado ao host via rota ***ToHost*** e o processo continuará no estado ***BD_normal***.

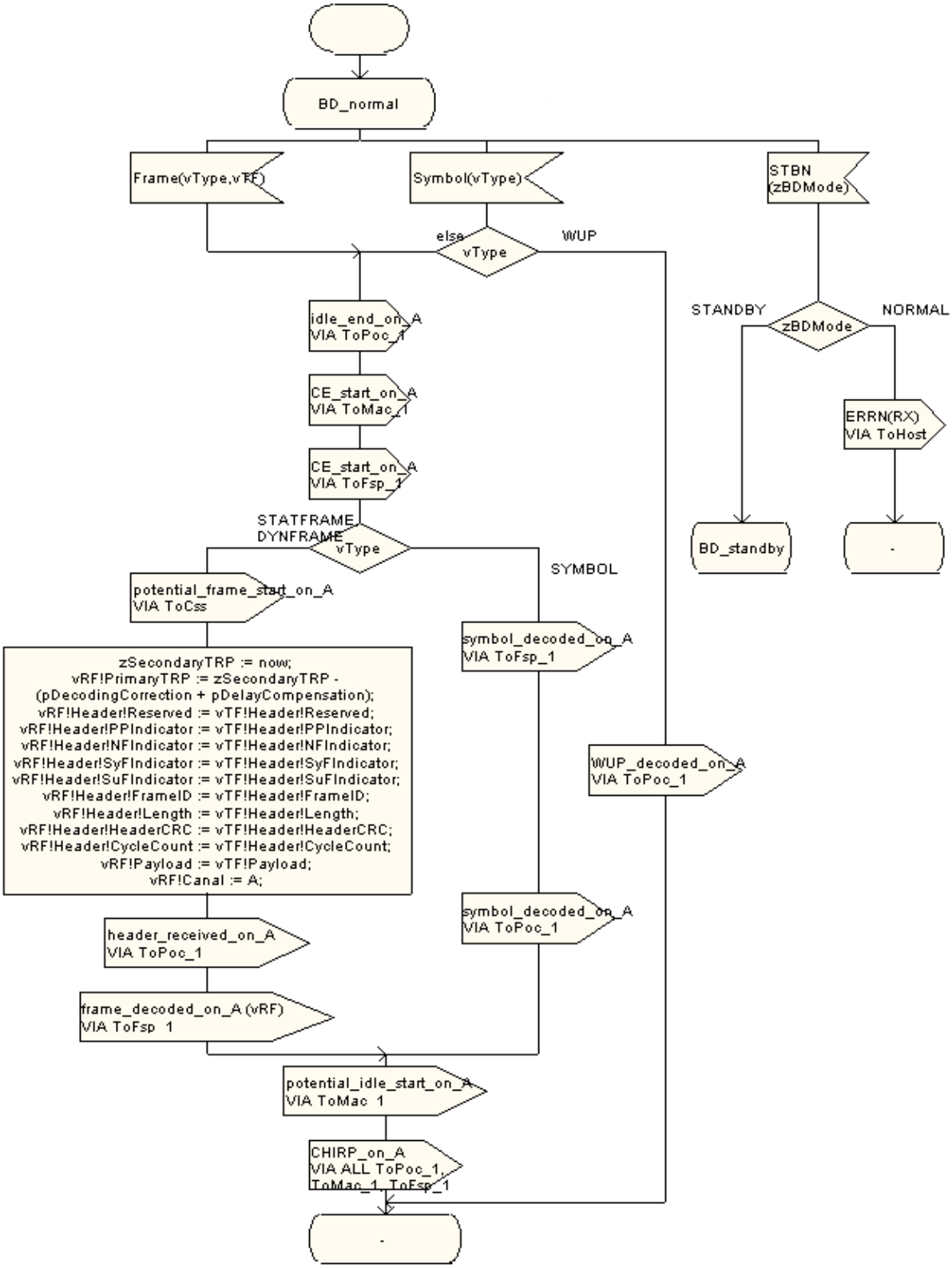


Figura 3.17: Especificação do processo receptor do controlador de barramento

3.3 Especificação em SDL do bloco Bus

Para simular uma rede completa fez-se necessário a especificação em SDL do meio físico dentro do sistema. Dessa forma o barramento foi implementado no bloco ***Bus*** garantindo o real funcionamento de uma rede FlexRay.

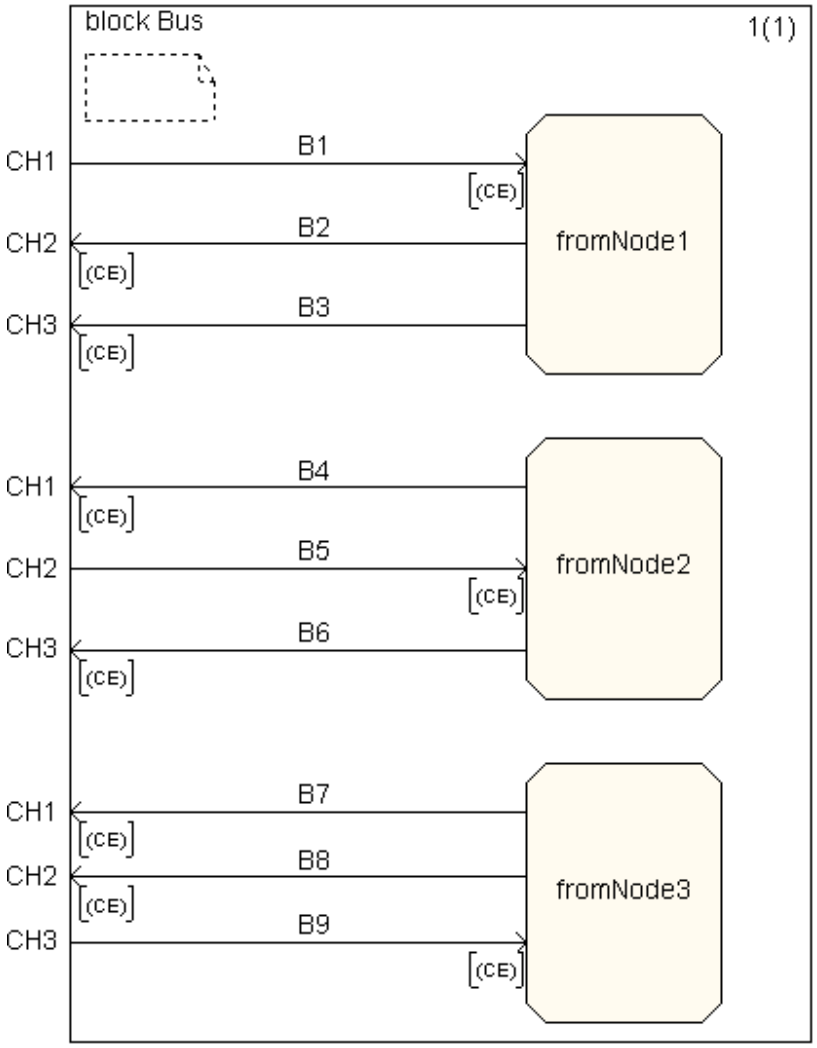


Figura 3.18: Bloco que representa o barramento da rede FlexRay

O único papel desse bloco é retransmitir os sinais recebidos de um nó para todos os outros nós. Três processos foram especificados para cumprir esse papel. Um processo poderia ter sido suficiente para desempenhar essa função, mas dessa forma poderiam ocorrer problemas de endereçamento. Isso porque em SDL nenhum recurso permite que um sinal recebido seja enviado para um processo em uma situação em que os dois lados são idênticos [22]. Teria sido necessário

armazenar os PIds (número de identificação de cada processo) além de outras alterações mais complexas. Uma solução simples era usar um processo para cada nó da rede.

A Figura 3.18 ilustra o bloco ***Bus*** contendo seus três processos: ***fromNode1***, ***fromNode2*** e ***fromNode3***. Cada processo é responsável pela recepção de sinal de um nó específico e retransmissão para todos os outros nós. Logo, o processo responsável pelo nó ***Node1***, ***fromNode1***, recebe um dos sinais especificados na lista de sinais ***CE*** (*frame* ou símbolo) provenientes do canal externo ***CH1*** e retransmite o mesmo sinal para os canais externos ***CH2*** e ***CH3***, canais que interligam o barramento aos nós ***Node2*** e ***Node3*** respectivamente. O mesmo procedimento é realizado para os processos ***fromNode2*** e ***fromNode3***, que são responsáveis pelos nós ***Node2*** e ***Node3*** respectivamente.

A Figura 3.19 ilustra o conteúdo do processo ***fromNode1***. O processo inicia-se em seu único estado, ***ready***. A partir de então ele aguarda o recebimento de um sinal para depois retransmiti-lo aos nós ***Node2*** (através da rota de sinal ***B2***) e ***Node3*** (através da rota de sinal ***B3***).

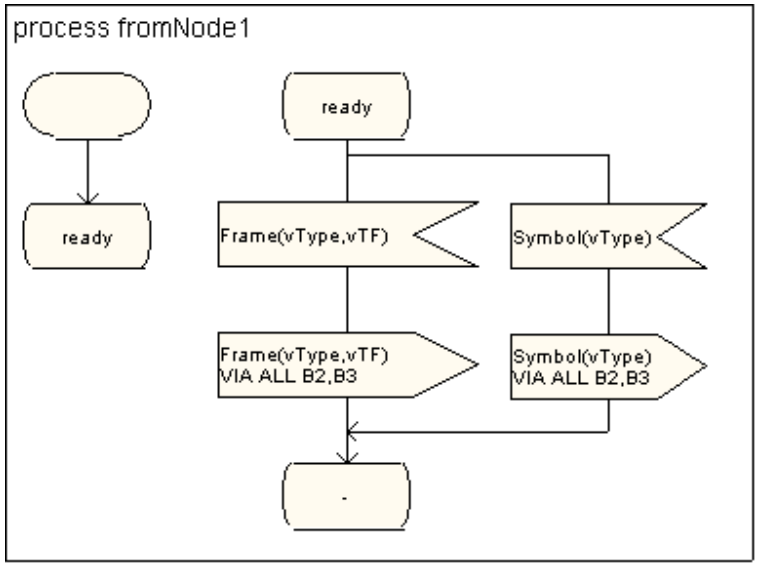


Figura 3.19: Processo responsável por retransmitir os sinais recebidos do nó 1 para todos nós da rede

3.4 Adaptações na especificação do protocolo FlexRay

Além da descrição completa da estrutura da rede, algumas adaptações também foram feitas em relação à especificação do FlexRay a fim de gerar um sistema executável usando SDL.

As principais adaptações foram:

- Substituição de todas as construções de macros descritas em [5] por construções de procedimentos.
- Criação do bloco controlador de barramento ***Bus_Driver***.
- Adaptação dos tipos existentes de temporizadores usados pelo protocolo FlexRay para um único tipo de temporizador baseado na unidade *microtick*.
- Substituição de alguns termos descritos em [5] por novos termos devido a conflito com palavras-chave utilizadas pela ferramenta SDL.
- Criação de um procedimento ***CycleCounter*** para verificar o ciclo de comunicação atual.
- Substituição de todas as construções "*task*", contendo textos informais relacionados ao *host*, por sinais de saída usando as construções "*output*".
- Substituição da construção "*task*" ***import_vTCHI_from_the_CHI*** no procedimento ***ASSEMBLE_STATIC_FRAME_A*** por um sinal de entrada recebido do CHI.

Alguns mecanismos do protocolo especificados em [5] usam construções de macros com o exclusivo propósito de simplificar a apresentação do protocolo em SDL. Porém, esta não é uma prática aconselhável na implementação de sistemas reais portáteis. Macros são geralmente dependentes de certas ferramentas SDL, logo o projetista poderia ter dificuldade ao tentar compilar seu modelo utilizando diferentes editores de SDL. Dessa forma, todas as construções que faziam uso de macros foram substituídas por construções de procedimentos. Tanto a construção macro quanto a construção procedimento (*procedure*) possuem basicamente a mesma funcionalidade. Não interferindo, assim, no modo de operação do protocolo FlexRay. A Figura 3.22 ilustra uma comparação da especificação da macro em [5] e da especificação do procedimento no projeto.

A criação do bloco controlador de barramento ***Bus_Driver***, como explicado na seção 3.2.2, foi fundamental na geração de um modelo SDL executável do sistema FlexRay, já que este componente não está especificado em [5]. Informações significantes para a implementação do

bloco ***Bus_Driver*** foram extraídas de [5] e a partir das informações coletadas o bloco ***Bus_Driver*** foi modelado.

A representação do tempo no protocolo FlexRay é baseada em uma hierarquia que inclui *microticks* e *macroticks*. Os *microticks* são múltiplos inteiros do período do *clock* local de cada nó, já o *macrotick* consiste de um número inteiro de *microticks* [18]. Vários mecanismos do FlexRay necessitam de temporizadores que medem certo número de *microticks* ou *macroticks*. A especificação do FlexRay usa uma extensão dos temporizadores do SDL para realizar tal função, porém esta extensão não é suportada pela maioria das ferramentas editoras de SDL. Os editores de SDL utilizam uma base de tempo única, conseqüentemente, podem utilizar apenas um tipo de temporizador. Porém, foi verificado que existe uma relação definida entre a unidade de tempo *microtick* (μT), a unidade de tempo *macrotick* (MT) e a unidade de tempo da amostragem de bits *sampletick* (ST). Dessa forma, para solucionar este problema a unidade *microtick* foi definida como sendo a unidade básica de tempo do sistema. Logo, todas as outras unidades de tempo (*macrotick* e *sampletick*) tiveram seus valores recalculados como segue: 1MT é igual a 40μT e 1ST é igual a 1μT. Assim, os temporizadores *sampletick* foram diretamente convertidos para temporizadores *microtick* e os temporizadores *macrotick* tiveram seus valores multiplicados por 40 (valor este representado pela constante ***pMicroPerMacroNom***). Um exemplo reside na declaração do temporizador ***tMinislot*** do processo ***MAC_A***: "***timer tMinislot := gdMinislot * pMicroPerMacroNom;***". Nesse exemplo o temporizador ***tMinislot*** que tem valor igual a 40MT (representado pela constante ***gdMinislot***) é declarado como sendo multiplicado pela constante ***pMicroPerMacroNom*** (número inteiro de *microticks* por cada *macrotick*).

Alguns termos descritos em [5] entram em conflito com as palavras-chave das ferramentas editoras de SDL, dentre elas: *ALL*, *STATE*, *CHANNEL* e *ACTIVE*. A solução encontrada foi substituir esses termos por outras palavras livres de conflito. Os termos conflitantes foram substituídos por: ***TODOS***, ***ESTADO***, ***CANAL*** e ***ATIVO***, respectivamente. Apesar desta adaptação não ter impacto no projeto sua descrição nesta seção visa apenas facilitar o entendimento do leitor ao analisar os diagramas desenvolvidos. A Figura 3.22 ilustra esta alteração.

As decisões de mudança dos modos de operação nos processos, feita pelo processo POC no final de cada ciclo de comunicação, dependem se o número do ciclo atual é par ou ímpar. Porém [5] não descreve nenhum mecanismo para verificar o estado do ciclo. Assim, foi criado um

procedimento chamado ***CycleCounter*** responsável por verificar o ciclo de comunicação atual durante a operação do FlexRay. Esse mecanismo de verificação do ciclo de comunicação é de suma importância para o funcionamento da rede FlexRay. A Figura 3.20 mostra como o procedimento foi especificado. O procedimento recebe como entrada um parâmetro do tipo inteiro, faz um processamento deste valor para verificar se é par ou ímpar e então retorna o resultado correspondente do tipo ***T_EvenOdd*** (tipo especificado em [5]).

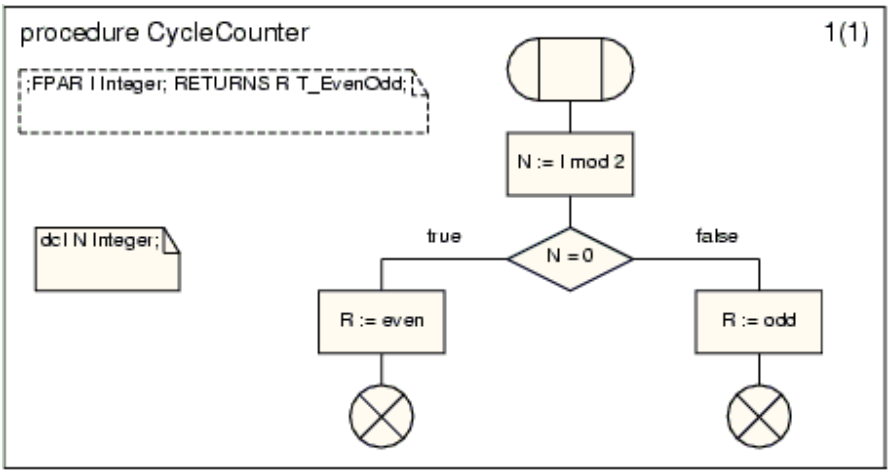


Figura 3.20: Procedimento que verifica o ciclo de comunicação do sistema

Para gerar um modelo SDL mais próximo de uma implementação real algumas construções "*task*", contendo textos informais relacionados à exportação de variáveis do CHI, foram substituídas por sinais de saída usando as construções "*output*". A Figura 3.21a mostra que no diagrama de [5] encontra-se uma construção "*task*" contendo dois comandos: o primeiro é uma atribuição de valor a uma variável e o segundo nada mais é do que um texto sem valor funcional (*update vPOC in CHI*); ao passo que no diagrama especificado neste trabalho, Figura 3.21b, ainda existe uma construção "*task*" referente à atribuição de valor a variável, porém o texto sem valor transformou-se em um sinal de saída (construção *output*) totalmente funcional que será enviado ao *host* para cumprir seu propósito.

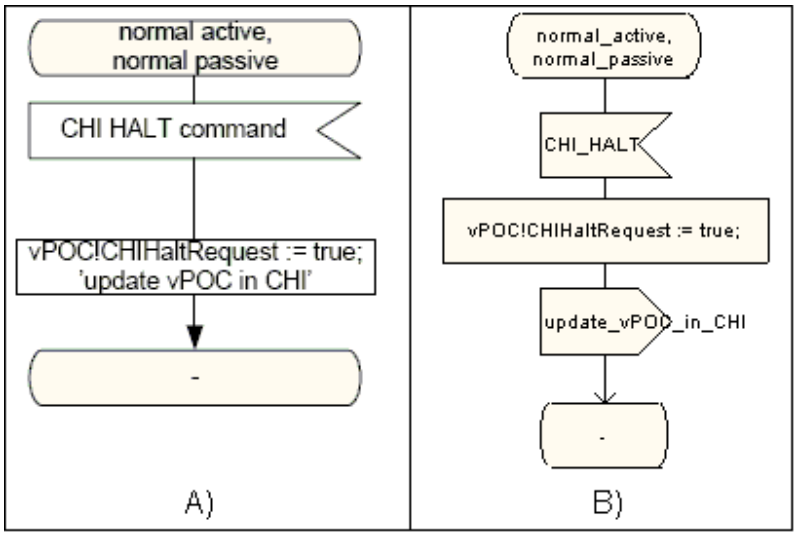


Figura 3.21: Construção *task* encontrada em [5] e construção *output* especificada neste trabalho

O mesmo tipo de alteração acontece no procedimento ***ASSEMBLE_STATIC_FRAME_A*** localizado no processo ***MAC_A***, mas ao contrário da alteração anterior uma construção "*input*" está substituindo a construção "*task*" original que contem texto sem valor funcional (***import_vTCHI_from_the_CHI)***. A Figura 3.22a mostra que no diagrama de [5] encontra-se uma construção "*task*" contendo um texto sem valor funcional (*import vTCHI from the CHI*); ao passo que no diagrama especificado neste trabalho, Figura 3.22b, no lugar da construção "*task*" agora existe uma construção de estado (*state*) e uma construção de "*input*" para entrada do sinal ***import_vTCHI_from_the_CHI(vTCHI)***. Então, o sistema pode receber dados do *host* e operar normalmente.

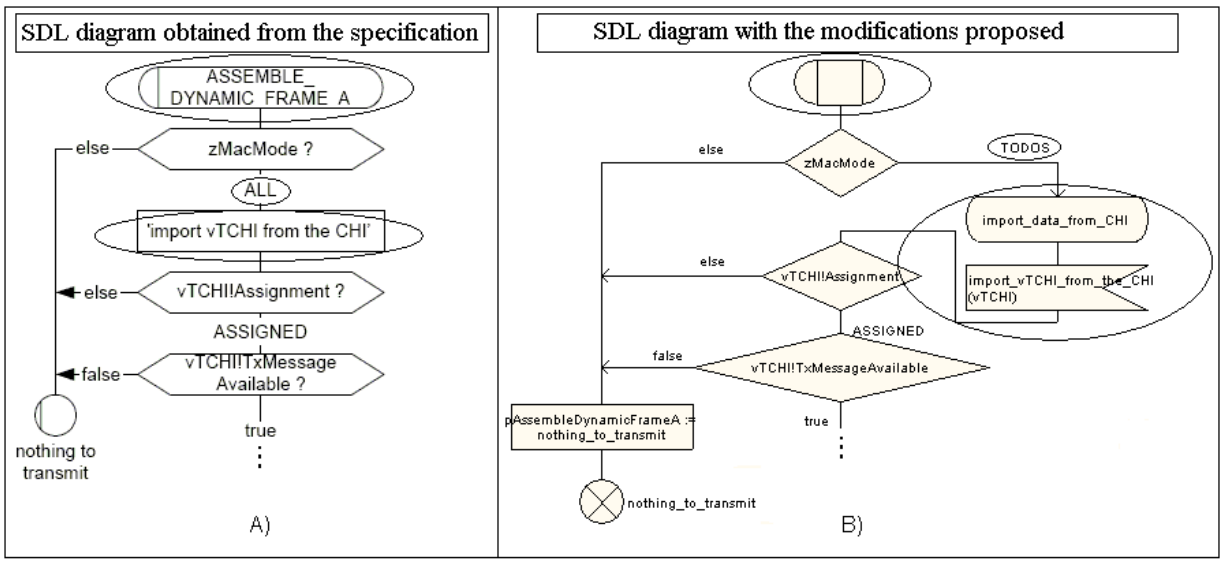


Figura 3.22: Diagrama SDL descrito em [5] e diagrama SDL especificado neste trabalho

Na Figura 3.22 estão ilustradas três das modificações mencionadas anteriormente e que foram feitas em [5] a fim de aprimorar o modelo SDL. Dentre as alterações propostas estão ilustradas nesta figura: a descrição do diagrama usando procedimento ao invés de macro (símbolos iniciais dos diagramas), a substituição do termo *ALL* por ***TODOS*** e a substituição da construção "*task*" *import vTCHI from the CHI* pelo sinal de entrada ***import_vTCHI_from_the_CHI (vTCHI)*** através da construção "*input*", onde ***vTCHI*** é a variável que armazena os dados enviados pelo *host* e que serão enviados pelo nó pelo barramento.

Estes aprimoramentos propostos nesta subseção permitiram que a rede FlexRay deste projeto operasse integralmente.

Capítulo 4

Resultados da Validação, da Simulação e Geração de Diagramas de Troca de Mensagens

A validação é um recurso disponibilizado pela ferramenta *Validator* do SDT para facilitar a identificação de erros cometidos durante o processo de especificação do sistema. Através da validação, todos os estados possíveis do sistema especificado são percorridos, verificando em cada estado os sinais que podem ser enviados/recebidos e se algum deles não está sendo tratado adequadamente. A validação possibilita o aumento da produtividade e qualidade no desenvolvimento de sistemas em SDL.

A simulação é utilizada para se verificar a dinâmica do sistema em casos especiais da especificação. É utilizada principalmente para se verificar, de forma mais detalhada, o comportamento do sistema em alguns casos críticos.

A validação e a simulação do sistema modelado neste trabalho foram realizadas através do *Validator* e *Simulator,* ferramentas integrantes do pacote SDT [12, 15].

Na seção 4.1 é apresentado o resultado da validação do sistema especificado no capítulo 3 e é analisado como o processo de validação pode ser utilizado para a identificação de erros de especificação. Na seção 4.2 são apresentadas algumas simulações visando exemplificar o funcionamento do sistema. Os resultados das simulações são mostrados através de gráficos MSC (*Message Sequence Charts*) que mostram detalhadamente o correto comportamento dos processos dentro de cada tipo de nó.

4.1. Validação

A validação é um mecanismo automatizado de detecção de falha que verifica a robustez da aplicação e procura inconsistências e problemas em uma fase inicial do desenvolvimento. Também é usado para verificar se a própria especificação formal tem lógica consistente. Os sistemas podem ter alguns padrões de falhas. As falhas mais importantes são [20]:

- *Deadlocks*: Estados em que nenhuma execução adicional é possível. Por exemplo, quando todos os processos estão esperando por condições que nunca poderão ser satisfeitas.
- *Livelocks*: Seqüências de execução que podem ser repetidas indefinidamente sem fazer progresso eficaz.
- Terminação imprópria: A conclusão de uma execução sem satisfazer a condição de término apropriada.

A validação de um sistema é baseada em uma técnica chamada exploração do espaço do estado, que é usada para a análise automática de sistemas distribuídos. O espaço de estados do sistema contém todos os estados alcançáveis do sistema e as maneiras possíveis para que eles possam ser alcançados. A estrutura do espaço de estados pode ser descrita através de árvores de comportamento ou gráficos de alcançabilidade como na Figura 4.1. Essas estruturas representam o comportamento de um sistema. Os nós da árvore representam estados do sistema. A soma de todos os estados do sistema é o espaço de estados. As transições representam eventos de SDL como atribuições, entrada e saída dos sinais e das tarefas.

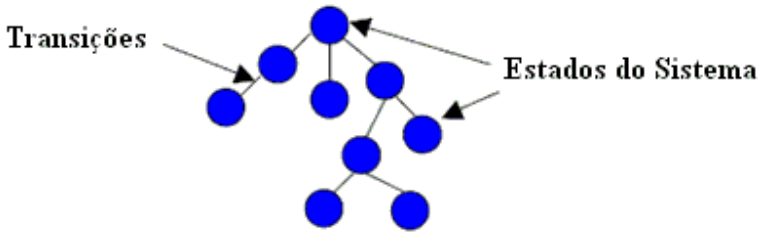


Figura 4.1: Árvore de comportamento do sistema

Há vários tipos de algoritmos de exploração do espaço de estados que poderiam ser aplicados para validar um sistema. O módulo *Validator* do SDT, responsável pela realização da validação das especificações em SDL deste trabalho, disponibiliza cinco formas diferentes de se percorrer os estados de um sistema durante o processo de validação: *Exaustive Exploration, Bit State Exploration, Random Walk, Tree Walk* e *Tree Search.*

O *Exaustive Exploration* é o algoritmo utilizado quando se deseja que todos os caminhos possíveis do espaço de estados do sistema sejam percorridos. Trata-se de um algoritmo de exploração utilizado para sistemas pequenos. À medida que o sistema cresce e o número de estados aumenta significativamente, a utilização deste algoritmo se torna impraticável.

O *Bit State Exploration* funciona de forma semelhante, entretanto cada estado é representado através de uma forma mais compacta, utilizando uma estrutura de dados chamada

hashtable para gerenciar os estados percorridos. Trata-se de um algoritmo mais eficiente para sistemas de maior complexidade. Entretanto, se o número de estados do sistema for realmente grande, este método pode também ser ineficaz.

O *Random Walk* é indicado para sistemas de maior porte, onde o número de estados é bastante significativo. Este algoritmo percorre de forma aleatória os ramos da árvore com os estados possíveis do sistema, simplificando bastante o gerenciamento de estados percorridos e não percorridos. Este método é apropriado para sistemas excepcionalmente grandes.

O *Tree Walk* é baseado em um algoritmo fortemente determinístico que executa uma seqüência de buscas pela árvore de estados aumentando a profundidade e começando em estados diferentes no espaço de estados. Sempre que o algoritmo faz uma transição que aumente a cobertura, o estado seguinte é armazenado em uma lista de estados de teste que são usados para futuras buscas pela árvore. Este método é apropriado para todos os tipos de sistemas.

O *Tree Search* executa uma busca pela árvore de estados a partir do estado atual do sistema. Este algoritmo realiza uma exploração de forma que todas as combinações de ações possíveis sejam executadas. Este método não é apropriado para grandes sistemas.

Todos os algoritmos executam uma exploração automática do espaço de estados que parte do estado atual do sistema.

4.1.1. A validação do sistema SDL especificado

O *Tree Walk* foi o método escolhido para validar o sistema SDL desenvolvido neste trabalho, visto que o mesmo apresentou a melhor cobertura do espaço de estados do sistema. Para a execução da validação utilizando o método *Tree Walk*, alguns parâmetros são necessários:

• *Depth*: Indica a profundidade máxima da árvore de estados atingida pelo algoritmo;

• *Timeout*: Permite que o usuário defina um tempo limite para a finalização do processo de validação;

• *Coverage*: Permite que o usuário defina um limite para a porcentagem de símbolos SDL percorridos pela validação.

Os valores dos parâmetros utilizados durante a validação foram: *Depth* = 1000000, *Timeout* = 300 e *Coverage*: 100. Alguns parâmetros de menor valor foram testados, porém à medida que eram aumentados, o número de estados alcançados também aumentava. A partir destes valores

(*Depth* = 1000000, *Timeout* = 300 e *Coverage*: 100), mesmo que eles fossem aumentados, o mesmo número de estados seria percorrido.

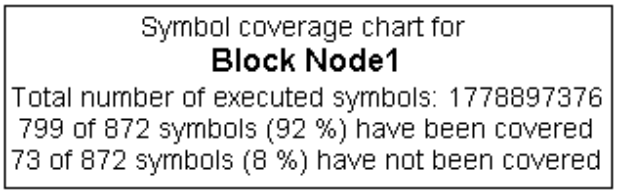


Figura 4.2: Cobertura de símbolos do Nó 1

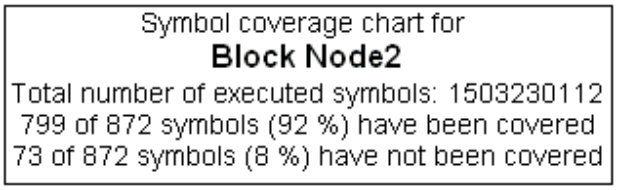


Figura 4.3: Cobertura de símbolos do Nó 2

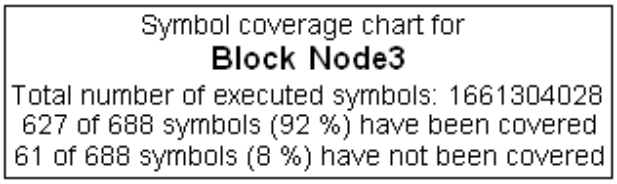


Figura 4.4: Cobertura de símbolos do Nó 3

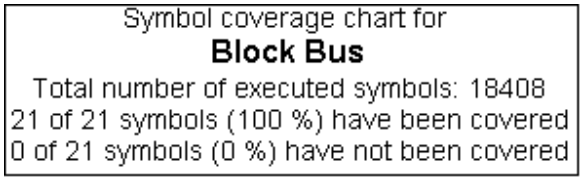


Figura 4.5: Cobertura de símbolos do bloco ***Bus*** (Figura 3.18)

As Figuras 4.2, 4.3, 4.4 e 4.5 mostram o número de símbolos cobertos e não-cobertos pela validação do sistema obtidos através da ferramenta *Coverage Viewer*, além de mostrar o número de vezes que os símbolos foram executados. Interpretando os resultados da validação observa-se que o nó 1 (***Node1***), o nó 2 (***Node2***) e o nó 3 (***Node3***) apresentam 92% dos símbolos percorridos e o bloco do barramento (***Bus***) apresenta 100% dos símbolos percorridos. O motivo para a diminuição da porcentagem de cobertura dos símbolos nos nós 1, 2 e 3 é a presença de ações condicionais na especificação, por exemplo, a condição de *if*. Apenas uma das condições, verdadeiro ou falso, será percorrida. O processo ***POC*** no bloco ***Communication_Controller*** contém vários pontos onde existem ações condicionais. Tais ações também interferem na

execução dos outros processos (MAC e FSP), diminuindo assim a quantidade de símbolos percorridos no nó.

Os símbolos não alcançados podem ser identificados através do *Coverage Viewer*, que exibe uma árvore com a hierarquia de blocos, processos, procedimentos e símbolos do sistema SDL, mostrando o que foi e o que não foi testado pela validação. A Figura 4.6 mostra o resultado do *Coverage Viewer* para a validação do sistema especificado neste trabalho relativo ao bloco *Bus*. Os processos e procedimentos preenchidos com cinza indicam que foram completamente testados. Caso os processos e procedimentos se encontrassem parcialmente preenchidos indicariam que alguns de seus símbolos não foram alcançados. Estes símbolos não alcançados podem ser identificados na especificação em SDL para que se possa verificar se há algo de errado ou se eles tratam casos inatingíveis pela validação.

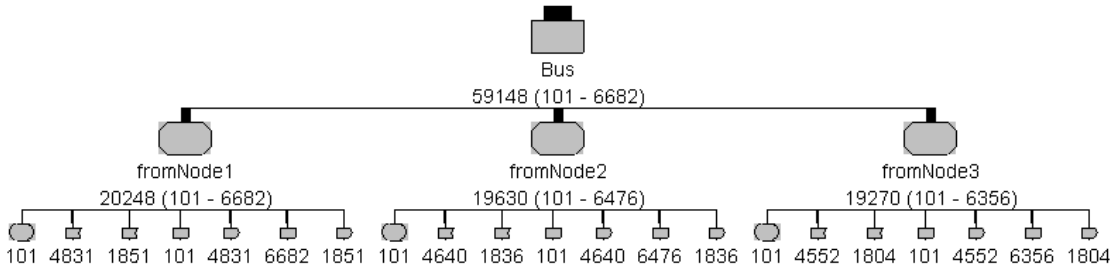


Figura 4.6: *Coverage Viewer*

Para que as porcentagens de cobertura dos símbolos nos nós 1, 2 e 3 (92,0%, 92,0% e 92,0%, respectivamente) fossem atingidas durante a validação, foi utilizado um recurso de não-determinismo da linguagem SDL para atribuir valores de teste aos parâmetros de entrada dos sinais que o ambiente envia ao sistema. Desta forma, diversas possibilidades diferentes foram testadas. A Figura 4.7 apresenta parte da definição de valores atribuídos que foram usados pelo processo de validação. Nesta figura está representada a definição dos valores do sinal ***SyncCalcResult***, para diversas possibilidades, enviado ao sistema FlexRay com destino ao processo ***POC*** durante o processo de validação.

```
Clear-Signal-Definitions SyncCalcResult
Define-Signal SyncCalcResult (WITHIN_BOUNDS, 0, false)
Define-Signal SyncCalcResult (WITHIN_BOUNDS, 1, false)
Define-Signal SyncCalcResult (WITHIN_BOUNDS, 2, false)
Define-Signal SyncCalcResult (WITHIN_BOUNDS, 0, true)
Define-Signal SyncCalcResult (WITHIN_BOUNDS, 1, true)
Define-Signal SyncCalcResult (WITHIN_BOUNDS, 2, true)
Define-Signal SyncCalcResult (EXCEEDS_BOUNDS, 0, false)
Define-Signal SyncCalcResult (EXCEEDS_BOUNDS, 1, false)
Define-Signal SyncCalcResult (EXCEEDS_BOUNDS, 2, false)
Define-Signal SyncCalcResult (EXCEEDS_BOUNDS, 0, true)
Define-Signal SyncCalcResult (EXCEEDS_BOUNDS, 1, true)
Define-Signal SyncCalcResult (EXCEEDS_BOUNDS, 2, true)
Define-Signal SyncCalcResult (MISSING_TERM, 0, false)
Define-Signal SyncCalcResult (MISSING_TERM, 1, false)
Define-Signal SyncCalcResult (MISSING_TERM, 2, false)
Define-Signal SyncCalcResult (MISSING_TERM, 2, true)
Define-Signal SyncCalcResult (MISSING_TERM, 1, true)
Define-Signal SyncCalcResult (MISSING_TERM, 0, true)
```

Figura 4.7: Definição dos valores de teste para o sinal ***SyncCalcResult***

4.1.2. Detecção de erros de especificação através da validação

Através do processo de validação, podem-se detectar erros de especificação que seriam dificilmente encontrados manualmente. Através do uso do *Coverage Viewer*, podem-se identificar os símbolos não alcançados pela validação e, assim, tem-se uma noção mais precisa dos pontos onde pode haver erros.

Um dos erros detectados foi a presença de um *deadlock* na modelagem do processo ***Transmitter*** do bloco ***Bus_Driver***. O erro foi encontrado após a execução da ferramenta *Validator*, analisando-se o resultado apresentado pelo *Coverage Viewer*. Durante a transmissão do frame do tipo ***DYNFRAME*** no processo ***Transmitter*** o sinal ***DTS_start_on_A*** é enviado para o processo MAC. No processo MAC esse sinal será consumido no estado ***wait_for_the_DTS_start*** e depois entrará no estado ***wait_for_the_AP_transmission_end*** onde receberá o sinal ***tMinislotActionPoint*** e assim enviará para o ***Transmitter*** o sinal ***stop_transmission_on_A***. Porém, caso o sinal ***DTS_start_on_A*** se perca ou ocorra algum tipo de falha no estado ***wait_for_the_DTS_start*** ou ***wait_for_the_AP_transmission_end*** o processo ***Transmitter*** nunca sairá do estado ***wait_for_DTS_end***. Para solucionar este problema foi adicionado um *input* para o sinal ***STBN*** no estado ***wait_for_DTS_end*** do processo ***Transmitter***.

A Figura 4.8 mostra o símbolo de *input* adicionado ao processo transmissor. Assim, caso haja problema na transmissão, o *host* poderá mandar o processo ***Transmitter*** para o estado ***BD_standby***, impedindo um travamento geral do sistema.

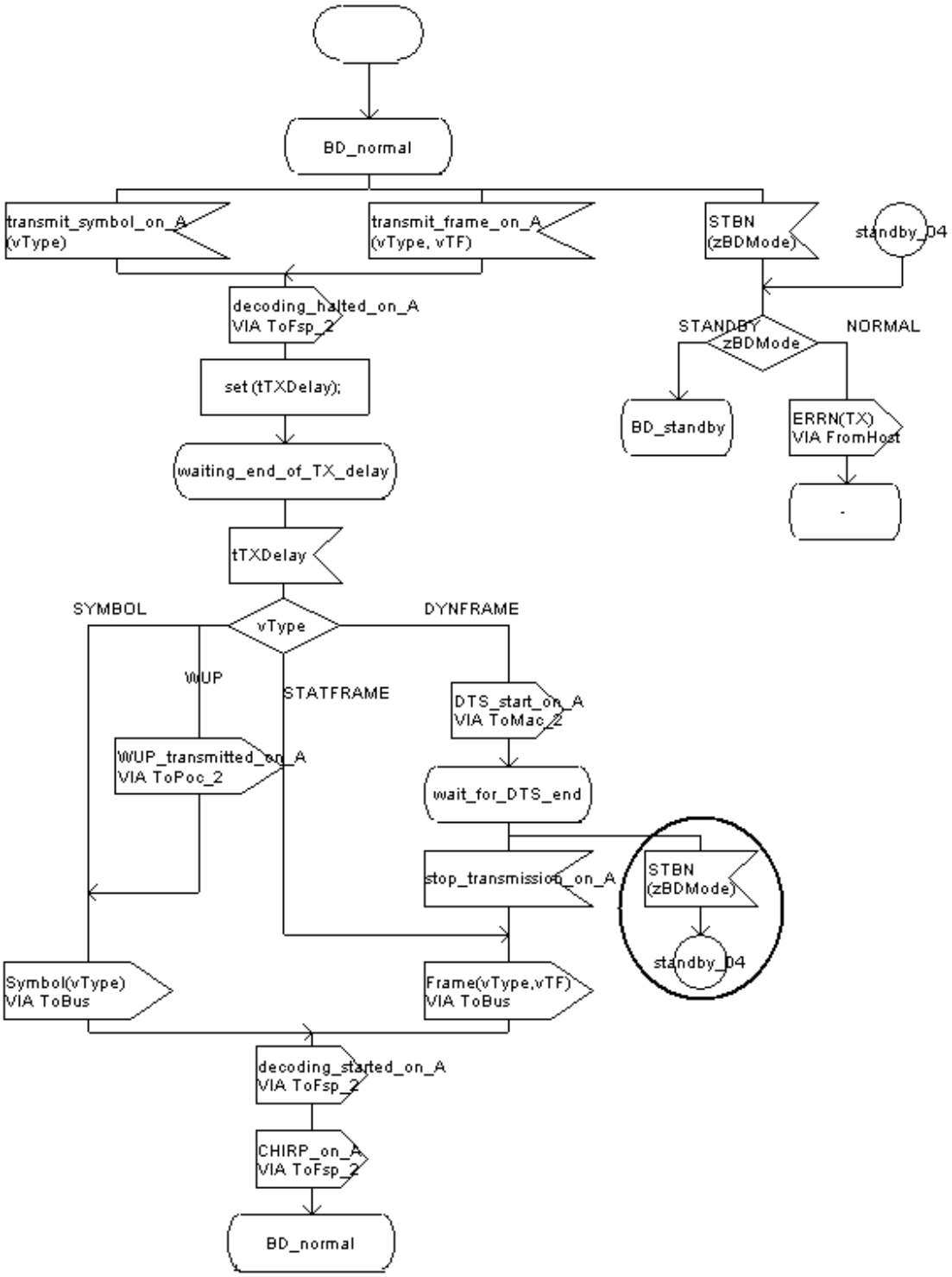


Figura 4.8: Símbolo de entrada de sinal ***STBN*** adicionado ao processo ***Transmitter***

Outro erro bastante comum, detectado pela validação, é o consumo implícito de sinal (*implicit signal consumption*). Durante a especificação do sistema, caso um sinal declarado não seja tratado pelo processo no estado em que ele se encontra, a validação informa a ocorrência deste erro. Durante a validação foi detectado um erro de consumo implícito de sinal no procedimento ***WAKEUP_LISTEN***. Este problema aconteceu porque foram ativados dois temporizadores e quando um deles expirou, seu sinal foi consumido, ao passo que o sinal proveniente do outro temporizador foi descartado. Tal erro foi facilmente descoberto depois da execução da ferramenta *Validator*, notando-se que não havia um comando para desativar o segundo temporizador depois do término do primeiro. A solução foi implementar um sistema para verificação e desativação do segundo temporizador. A Figura 4.9 ilustra a solução adotada.

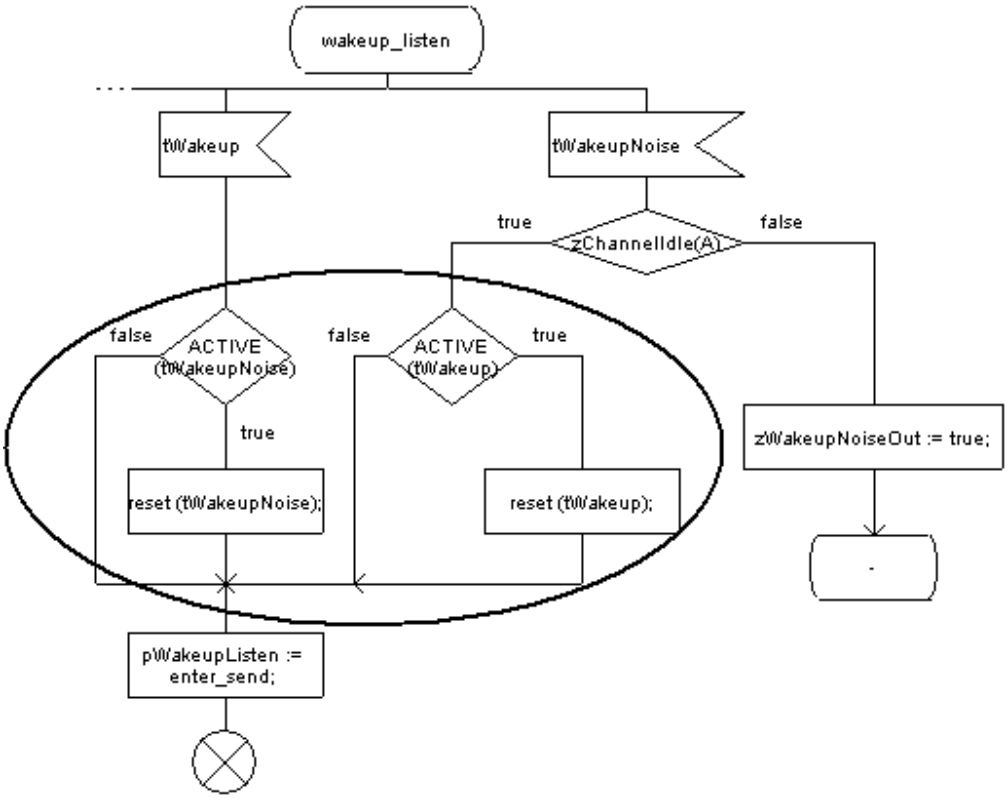


Figura 4.9: Sistema adicionado para evitar sinal implícito

Outros tipos de erros também puderam ser detectados durante a validação. Como é o caso do envio de sinais a processos que não existem mais. Caso uma referência a um processo que já foi destruído continue sendo armazenada em outro processo, um sinal pode ser enviado para uma

referência (*Pid*) inexistente. Para evitar isso, sempre que um processo é destruído todos os processos que possam possuir referências para ele devem ser notificados.

Após a correção dos erros detectados pelo processo de validação, o sistema ainda pode ser submetido a testes mais detalhados visando à verificação da existência de erros de lógica, que acarretam no funcionamento inadequado de algum procedimento especificado.

Pode-se então utilizar do recurso de simulação para a realização de testes de eventuais casos críticos e das principais funcionalidades do sistema, analisando-se de forma mais criteriosa cada passo de sua execução.

4.2. Simulações

Uma das vantagens mais significantes do SDL é a possibilidade do sistema ser simulado facilmente nas fases de projeto. Conseqüentemente, erros e ambigüidades podem ser detectados e corrigidos antes da fase de implementação, aumentando a qualidade do modelo e diminuindo consideravelmente o tempo e custo do projeto.

Após a completa validação do sistema, foram simuladas as principais funcionalidades do sistema FlexRay para a garantia da validade do sistema. Para tanto foi utilizada a ferramenta *Simulator*, que também é parte integrante do pacote *SDT*. Esta ferramenta permite o acompanhamento total da simulação gerando diagramas de trocas de mensagens (MSCs), que permitem visualizar a interação entre os diversos componentes do sistema.

Para a simulação do modelo FlexRay proposto foram utilizados os seguintes cenários:

- Cenário 1: processo de inicialização de um nó *coldstart* líder.
- Cenário 2: processo de inicialização de um nó *coldstart* seguidor.
- Cenário 3: processo de inicialização de um nó não-*coldstart*.
- Cenário 4: a inicialização da rede é finalizada após todos os nós estarem integrados à rede funcionando normalmente.
- Condições de erro em presença de interferência de ruído:
  - Cenário 5: o *host* induz comandos para interromper a comunicação imediatamente.
  - Cenário 6: condições de erro fatal detectadas pelos processos POC, MAC ou FSP.

O processo de inicialização da rede é chamado de *coldstart*. Somente alguns nós estão autorizados a inicializar a rede, chamados nós *coldstart*. Um nó *coldstart* que efetivamente inicia a rede é chamado de *leading coldstart node* (nó *coldstart* líder) e um nó que se integra à rede após outro nó *coldstart* é chamado *following coldstart node* (nó *coldstart* seguidor). De acordo

com [5], três diferentes tipos de caminhos podem ser seguidos pelo nó dependendo de sua configuração. O primeiro caminho refere-se ao nó *coldstart* líder, o segundo caminho refere-se ao nó *coldstart* seguidor e o terceiro caminho refere-se ao nó que não é do tipo *coldstart*. A partir desta descrição foram gerados MSCs mostrando detalhadamente o correto comportamento dos mecanismos do protocolo dentro de cada tipo de nó. Para cada tipo de caminho três cenários diferentes foram simulados, dependendo da verificação de erro executada ao final de cada ciclo de comunicação. Dessa forma as principais situações que podem ocorrer durante a inicialização e a operação normal de uma rede FlexRay foram simuladas e descritas em forma de diagramas. Tais diagramas podem ser utilizados na criação de pacotes de testes para implementação de qualquer aplicação usando o FlexRay. Estes pacotes de testes podem ser produzidos por ferramentas específicas baseadas na notação TTCN (*Tree and Tabular Combined Notation*) [21], parte integrante do SDT, com o intuito de automatizar o processo de testes.

As Figuras 4.10, 4.11 e 4.12 ilustram parte dos diagramas MSCs gerados para os cenários 1, 2 e 3. Tais figuras contemplam, principalmente, a fase de *wakeup* da rede. A Figura 4.13 ilustra parte do diagrama MSC gerado para o cenário 4 e contempla o funcionamento normal da rede. As Figuras 4.14 e 4.15 ilustram parte dos diagramas gerados para cenários 5 e 6 com condições de erro na presença de ruído.

Wakeup é o nome da fase de ativação dos nós na rede. Para que isso ocorra é necessário que todos os nós estejam ligados e conectados ao canal. Apesar de vários nós poderem tentar ativar o canal da rede somente um pode fazê-lo.

Os diagramas das Figuras 4.10, 4.11 e 4.12 apresentam comportamento inicial similar, visto que representam o processo de inicialização de um nó na rede. A diferença se dá a partir do momento em que o processo POC alcança o estado ***ready***.

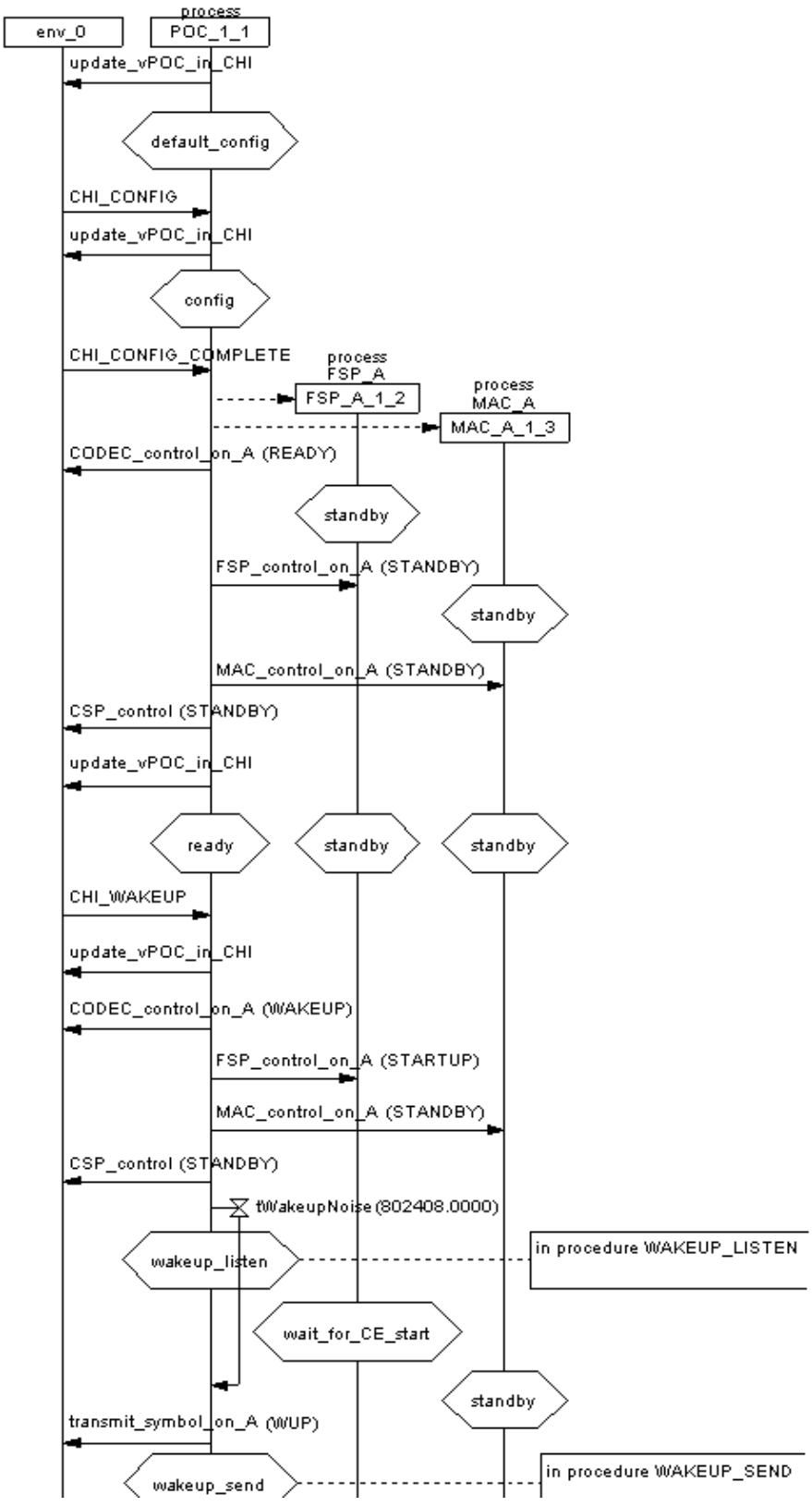


Figura 4.10: Cenário 1

O diagrama da Figura 4.10 refere-se à simulação do cenário de um nó *coldstart* líder (Cenário 1). Antes de entrar no estado ***default_config*** o processo POC inicializa o valor de suas variáveis. Após isso, ele aguarda um sinal do *host* para entrar no estado ***config*** e assim habilitar a configuração da rede.

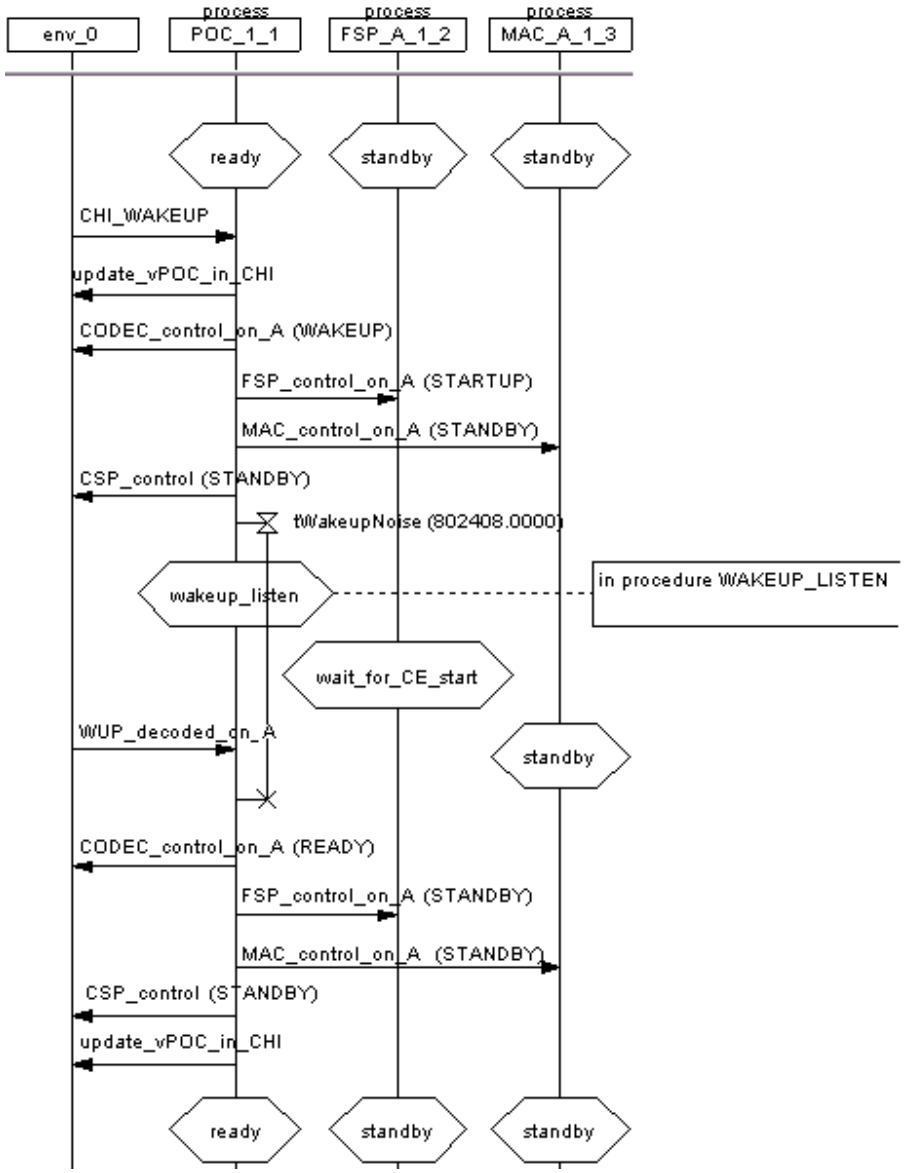


Figura 4.11: Cenário 2

Ao receber o sinal ***CHI_CONFIG_COMPLETE*** o processo POC cria todos os outros processos principais do protocolo, passa para o estado ***ready*** e aguarda a fase de *wakeup* da rede. O *host*, então, envia um comando ***CHI_WAKEUP*** ao POC autorizando sua entrada no procedimento ***WAKEUP***. No início deste procedimento são enviados sinais aos processos para

mudança de modo de operação. Em seguida um temporizador é ativado e o estado ***wakeup_listen*** é alcançado. Neste estado o nó aguarda a chegada de algum sinal proveniente de outro nó (caso haja comunicação em andamento) ou proveniente do temporizador que expirou (caso não tenha havido tentativas de ativação da rede). Como o temporizador expira, o nó deduz ser o único a tentar ativar a rede e então transmite o sinal responsável pela ativação dos nós, ***transmit_symbol_on_A(WUP)***.

O diagrama da Figura 4.11 refere-se à simulação do cenário de um nó coldstart seguidor (Cenário 2). Da mesma forma como ocorre ao nó *coldstart* líder, o nó *coldstart* seguidor alcança o estado ***wakeup_listen***. Neste momento ele recebe o sinal ***WUP_decoded_on_A***, o que significa que outro nó já iniciou a fase de ativação da rede antes dele. Diante disso o temporizador é desativado e a tentativa de ativação é abortada, fazendo com que o nó retorne ao estado ***ready*** do POC.

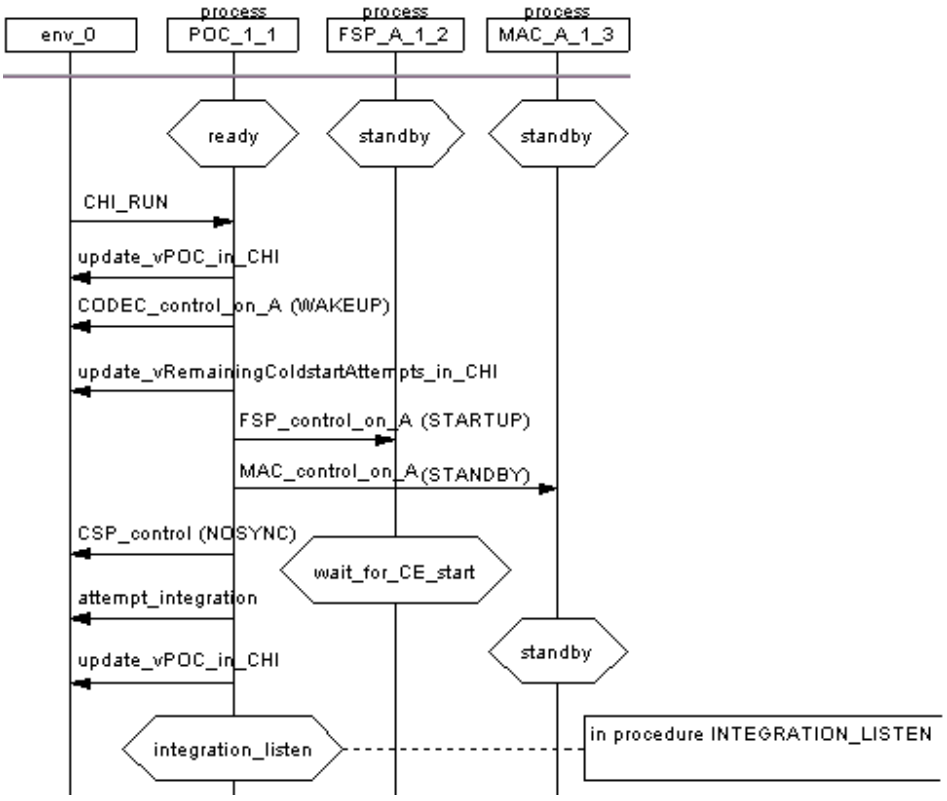


Figura 4.12: Cenário 3

Na Figura 4.12 é mostrado o cenário de um nó não-*coldstart* integrando-se à uma rede em funcionamento (Cenário 3). Este tipo de nó não possui o procedimento ***WAKEUP*** implementado

no processo POC, somente o procedimento ***STARTUP*** está disponível. Sendo assim, ao alcançar o estado ***ready*** o processo POC recebe o comando ***CHI_RUN*** do *host* para iniciar a fase de *startup* da rede. A partir daí o processo POC realiza uma série de tarefas e passa por vários estados dentro do procedimento ***STARTUP***.

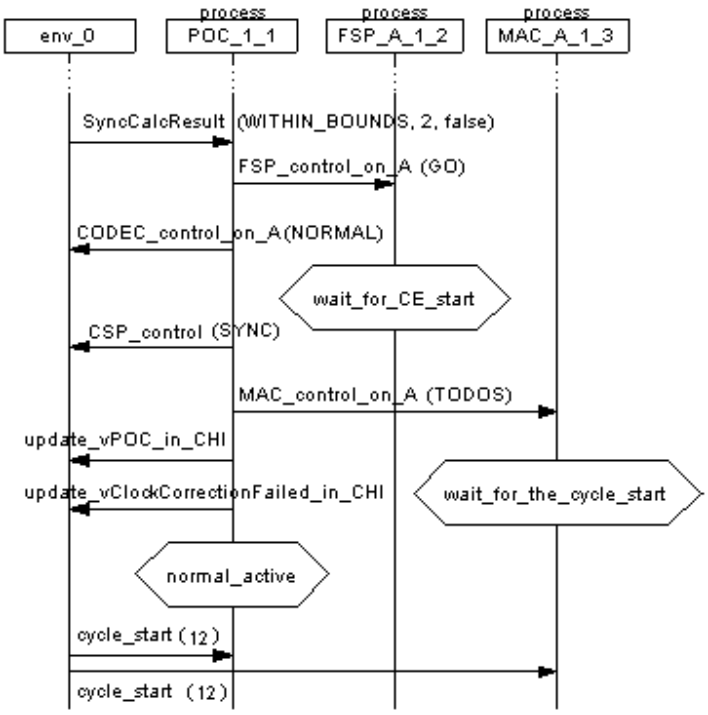


Figura 4.13: Cenário 4

A Figura 4.13 mostra o cenário onde a inicialização da rede é finalizada e todos os nós estão integrados à rede funcionando normalmente (Cenário 4). Após uma inicialização bem sucedida o processo POC irá permanecer no estado ***normal_active***. A partir de então ele executa uma seqüência de tarefas e no final de cada ciclo de comunicação o processo CSP envia o sinal ***SyncCalcResult*** ao POC para informar se o nó está sincronizado com a rede. Como o sinal informa o valor ***WITHIN_BOUNDS*** (dentro dos limites permitidos), então o POC envia sinais aos processos do controlador de comunicação informando-lhes o modo de operação que devem seguir. Por fim, o POC volta ao estado ***normal_active*** e inicia um novo ciclo de comunicação através do recebimento so sinal ***cycle_start***.

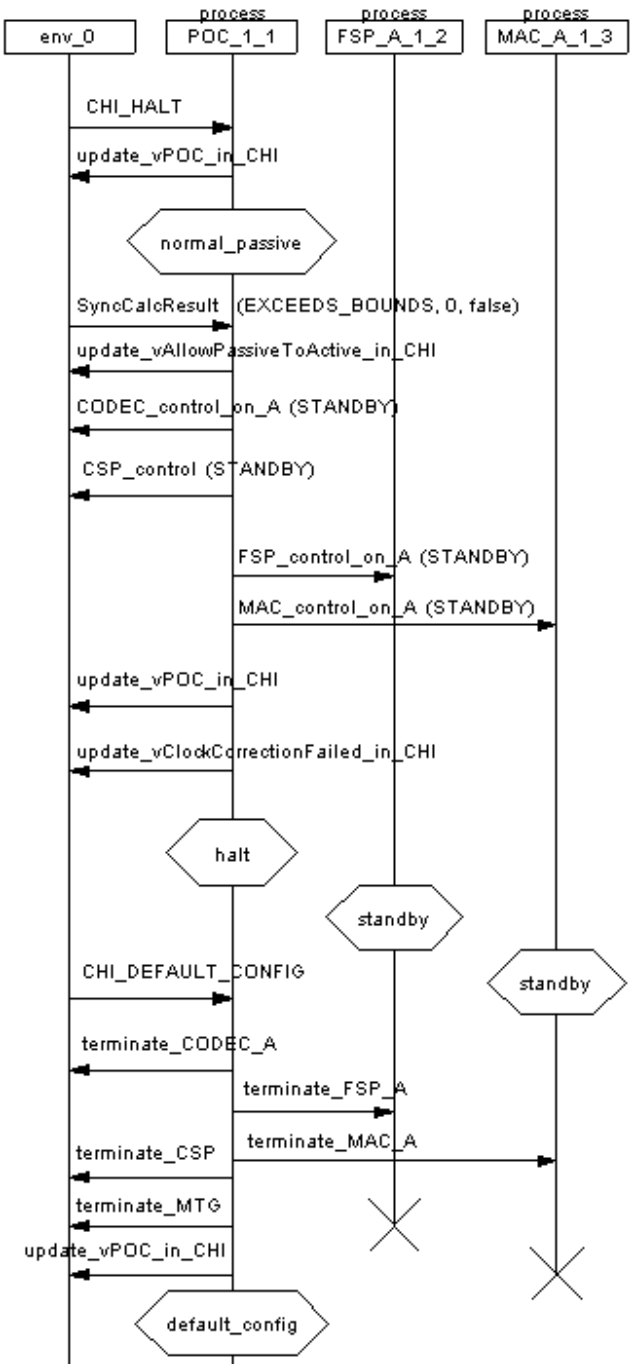


Figura 4.14: Cenário 5

O diagrama da Figura 4.14 simula um cenário onde ocorre reincidência de falha em um nó de uma rede em funcionamento (Cenário 5). O protocolo FlexRay possui um forte sistema tolerante a falhas. Dessa forma, em caso de verificação de falha, o nó não é forçado a uma interrupção brusca de operação, mas sim é enviado a um estado de observação (estado ***normal_passive*** do processo POC) onde será avaliada a natureza e amplitude do erro registrado.

O nó do diagrama encontra-se no estado ***normal_passive*** por já ter acusado erro de operação. Diante disso o *host* envia ao POC um comando ***CHI_HALT***, que autoriza uma interrupção da operação do nó caso a falha ocorra novamente. Ao final do ciclo de comunicação é verificada mais uma falha grave no sincronismo do nó através do recebimento do sinal ***SyncCalcResult*** com o valor ***EXCEEDS_BOUNDS*** (fora dos limites permitidos). A partir de então o POC passa para o estado ***halt***. Nesse estado assume-se que a recuperação para o estado ***normal_active*** (modo de operação normal) não pode ser alcançada, de modo que o POC paralisa os outros processos em preparação para a reinicialização do nó.

A Figura 4.15 simula o cenário onde acontece uma condição de erro causada por interferência de ruído (Cenário 6). O nó 1 envia o sinal ***transmit_frame_on_A*** para o controlador de barramento ordenando a transmissão de um *frame* depois de receber do *host* um sinal (***import_vTCHI_from_the_CHI***) que contém a mensagem a ser enviada no *frame*. Em seguida o controlador do barramento do nó 1 suspende o processo de decodificação através do envio do sinal ***decoding_halted_on_A*** e envia o sinal ***Frame*** para o barramento. O barramento, então, se encarrega de transmitir o sinal ***Frame*** aos outros nós conectados a ele. Ao receber o *frame* o controlador de barramento de cada nó dá início ao processo de decodificação. Quando o processo de transmissão é finalizado, o controlador de barramento do nó 1 envia o sinal ***decoding_started_on_A*** para seu controlador de comunicação ordenando o reinício do processo de decodificação. Porém, o sinal se perde devido à presença de interferência de ruído e não alcança o controlador de comunicação. A conseqüência é a ocorrência de um erro de *slot boundary crossed*, ou seja, o intervalo de tempo reservado à transmissão de *frame* concedida ao nó foi ultrapassado. Na ocorrência da ultrapassagem do limite de tempo de transmissão ou da ultrapassagem do limite de um dos quatro segmentos de comunicação (*static*, *dynamic*, *symbol* or nit) um sinal de erro fatal é enviado ao processo POC. Quando um erro grave ocorre, o POC é notificado para interromper a operação do controlador de comunicação do nó. Sendo assim, o POC envia um comando a cada processo configurando-os para o modo ***STANDBY*** e envia sinais ao *host* atualizando-o sobre o estado do nó.

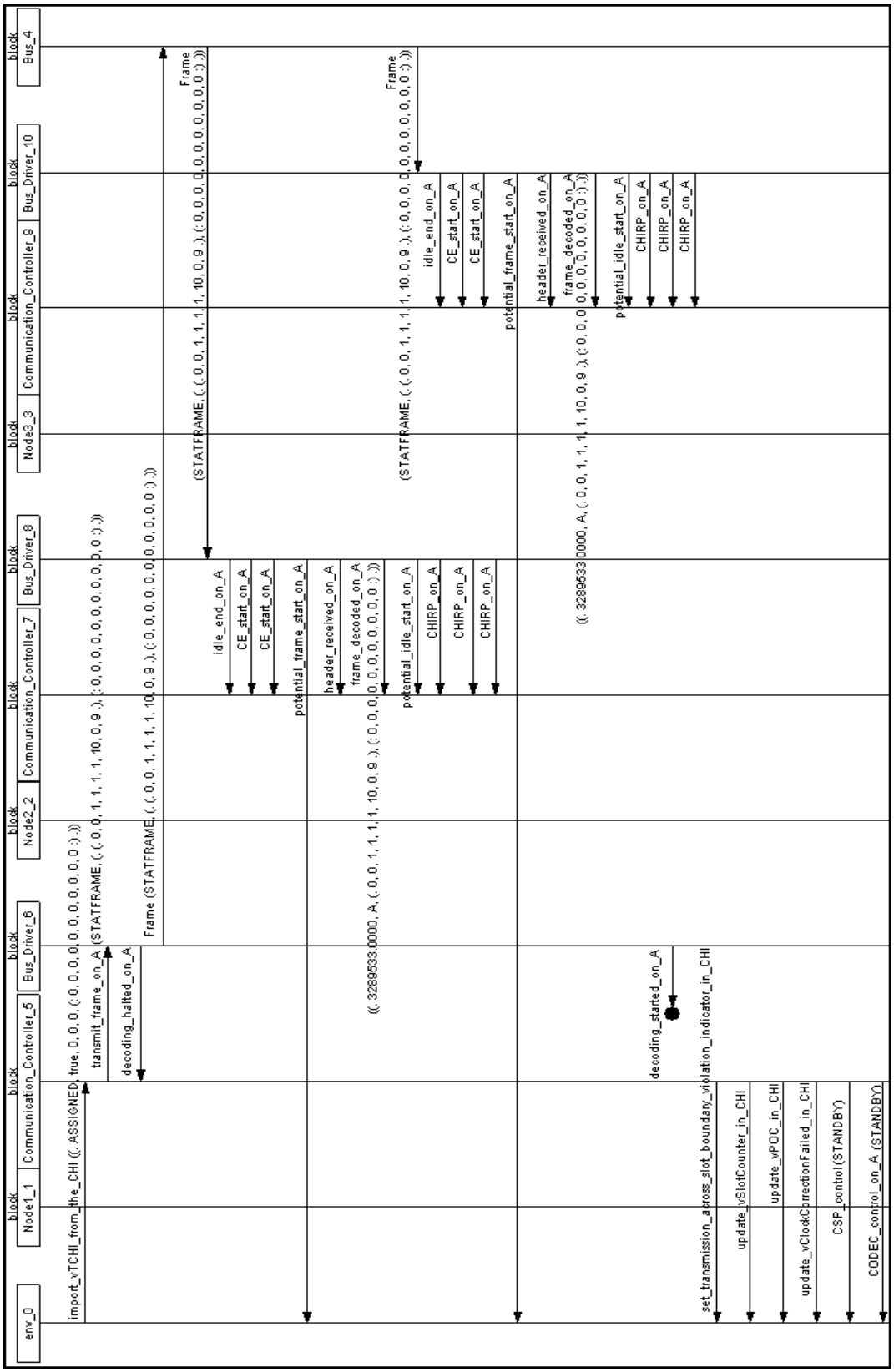


Figura 4.15: Cenário 6

Capítulo 5

Geração de código Java para sistemas de comunicação especificados em SDL

Após o processo de validação e simulação das principais funcionalidades e casos críticos do sistema desenvolvido em SDL, pode-se desenvolver uma implementação a partir de sua especificação formal.

Visando automatizar o processo de prototipagem de um sistema FlexRay, foi desenvolvida uma ferramenta capaz de gerar código em Java a partir da especificação do sistema em SDL. A utilidade da ferramenta desenvolvida não se atem apenas a geração de código para sistemas FlexRay, e sim atende a qualquer sistema de comunicação especificado em SDL usando as regras de mapeamento da Tabela 5.1.

A linguagem Java foi escolhida dentre as demais por diversas razões, contudo duas delas se destacam: a independência de plataformas e a facilidade de criação de interfaces gráficas. Por ser uma linguagem interpretada, o Java pode ser executado em qualquer plataforma ou equipamento que possua um interpretador Java. Além da portabilidade o Java provê interfaces gráficas com riqueza e sem a complexidade encontrada em outras linguagens de programação.

A seção 5.1 apresenta a tabela com as regras de mapeamento entre a linguagem SDL e Java utilizadas neste trabalho. A seção 5.2 apresenta a arquitetura da ferramenta desenvolvida focando na sua estrutura interna e no seu funcionamento. A seção 5.3 descreve o modelo de comunicação utilizado entre os objetos Java gerados pela ferramenta. Na seção 5.4 são exibidas propostas para as conversões das principais construções SDL que não possuem equivalentes diretos em Java. E, finalizando, na seção 5.5 é apresentada a interface gráfica utilizada pelo programa Java gerado para possibilitar o envio de sinais do usuário para o sistema e do sistema para o usuário.

5.1. Mapeamento da linguagem SDL para Java

A tecnologia Java se tornou uma importante força estimulando a evolução dos sistemas embarcados. A simplicidade e a independência de plataforma da linguagem atraíram desenvolvedores que se debatiam com problemas de portabilidade e prazos de entrega. Desde esse tempo, os programadores de dispositivos embarcados começaram a compreender e apreciar os benefícios da plataforma Java [22].

Para se criar uma ferramenta conversora de SDL para Java foi necessária a utilização de regras de mapeamento. Cada elemento sintático SDL corresponde a um elemento Java do mesmo nível sintático. A Tabela 5.1 apresenta as regras de mapeamento utilizadas neste trabalho.

Tabela 5.1: Regras de Mapeamento de SDL para Java

Entidade SDL	**Estrutura Java Equivalente**
Bloco	*Package*
Processo	Classe representando cada processo
Sinal	Método publico
Procedimento	Método privado
NewType	Classe representando tipo de dado correspondente
Syntype	Classe representando tipo de dado correspondente
Synonym	Atributo de interface (*public static*)
Timer	Classe com funcionalidades de *timer*

Os blocos são mapeados em Java *packages* (estrutura do Java de agrupamento de classes) contendo todos os tipos de dados definidos dentro do bloco. Os processos são mapeados em classes correspondentes colocadas dentro de seu *package* (bloco) correspondente. Os procedimentos são mapeados em métodos privados colocados dentro de sua classe (processo) correspondente.

As variáveis de temporizador foram mapeadas em classes *timer* do Java que apresentam a mesma funcionalidade da entidade *timer* do SDL. Dessa forma, comandos como *set* e *reset* podem ser utilizados, o estado do *timer* pode ser verificado (ativado ou desativado) e quando o tempo definido é atingido, eles chamam o método correspondente (o que é equivalente a enviar um sinal em SDL) como uma *thread* (chamada assíncrona).

Todos os tipos de dados em SDL foram mapeados para seu tipo de dado equivalente em Java. Os tipos de dados definidos fora dos blocos (no escopo da entidade *system* ou no escopo da entidade *package* do SDL) foram inseridos em um Java *package* especial chamado ***Declarations***.

5.2. Arquitetura da ferramenta geradora de código Java

A estrutura da ferramenta geradora de código Java desenvolvida neste trabalho foi baseada na arquitetura da ferramenta desenvolvida em [23]. A ferramenta geradora de código SDL para Java foi criada com o auxílio do JavaCC [16] (*Java Compiler Compiler*), que é uma ferramenta construtora de compiladores. Para isso, foi utilizado um programa que continha informações acerca das regras de formação da linguagem SDL. Para cada elemento da linguagem SDL (bloco, processo, etc) reconhecido em uma especificação que esteja sendo analisada, a ferramenta toma decisões sobre o que deve ser executado para converter a especificação em SDL. No apêndice B são apresentados conceitos relacionados com a construção de compiladores e o uso da ferramenta JavaCC.

O gerador Java desenvolvido recebe como entrada um arquivo com as especificações SDL na forma textual (SDL-PR) e gera na saída arquivos que contêm código Java. Dentro do gerador o arquivo com as especificações SDL passa por duas fases de reconhecimento (Figura 5.1). Na primeira fase é reconhecida a hierarquia dos blocos, processos e procedimentos. As informações coletadas são armazenadas em classes Java, sob a forma de árvore, responsáveis pelo armazenamento do sistema SDL de uma forma estruturada. São reconhecidos também os caminhos de comunicação existentes entre os processos de um mesmo bloco (*signal routes*) e entre os blocos do sistema (*channels*), através dos quais se podem identificar quais processos se comunicam entre si. Ainda nesta fase são reconhecidos os tipos de dados definidos no sistema (estruturas, conjuntos, enumerações, etc.).

Após o armazenamento em classes das informações obtidas através da fase 1, o gerador processa novamente o arquivo de entrada, reconhecendo desta vez as especificações comportamentais dos processos e procedimentos (fase 2). Tais especificações são convertidas para seu equivalente em Java e armazenadas nas respectivas classes que compõem a árvore de hierarquia construída na fase 1. Para a conversão das especificações comportamentais em SDL para o código Java equivalente são utilizadas as regras de mapeamento de SDL para Java propostas na Tabela 5.1, além de algumas técnicas descritas neste capítulo.

Ao final da fase 2 a especificação do sistema em SDL já foi integralmente traduzida para a linguagem Java. A partir daí inicia-se a última fase do processo de geração do código Java (fase 3). Nesta fase são adicionadas implementações de código Java pré-definidos aos arquivos resultantes da fase 2. Primeiramente adiciona-se o código responsável pela comunicação entre os objetos Java do sistema (seção 5.3) e por fim adiciona-se código responsável pela implementação da interface gráfica do sistema (seção 5.5).

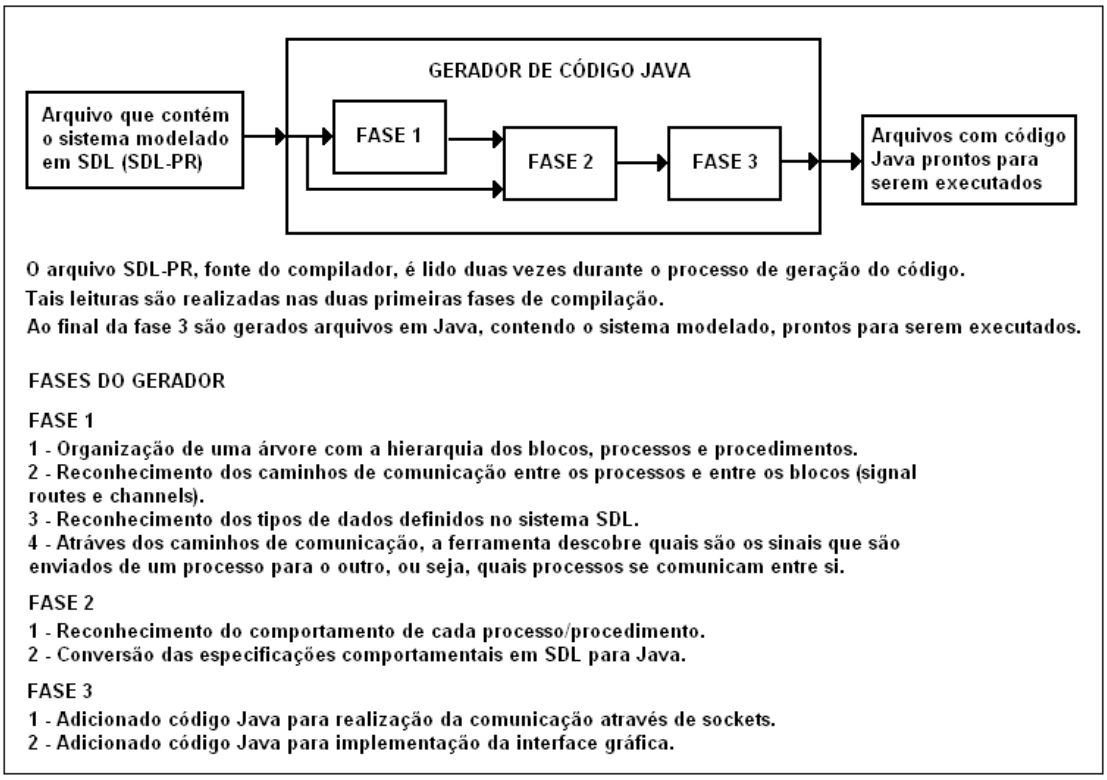


Figura 5.1: Ferramenta geradora de código Java

A ferramenta geradora de código Java desenvolvida é responsável por reconhecer e interpretar qualquer programa escrito em SDL (que siga a estrutura de rede proposta no capítulo 3) e convertê-lo para Java utilizando as regras de mapeamento de SDL para Java propostas na subseção 5.1.

A validação do código Java gerado foi feita manualmente. Para tanto o sistema foi executado tendo dados de teste como entrada e, então, seu comportamento operacional foi avaliado através dos resultados esperados na saída.

5.3. Modelo de comunicação entre os objetos Java gerados

Os blocos SDL são compostos por diversos processos que se comunicam entre si e também com processos de outros blocos. Para que a implementação em Java realizasse essa comunicação entre processos algum tipo de mecanismo deveria ser adotado. Existem várias tecnologias que se prestam a fornecer este tipo de mecanismo de comunicação, porém a mais eficiente e adequada para a finalidade deste trabalho refere-se ao *socket*.

O mecanismo de *socket* pode ser definido como um elo bidirecional de comunicação entre dois processos. Cada processo possui um endereço IP e uma porta única na rede. Dessa forma qualquer processo pode ser identificado.

Para a implementação da comunicação deste projeto foi utilizado o conceito de cliente/servidor por *socket* TCP/IP. Como o *socket* faz comunicação entre dois processos apenas, então para que um processo se comunique com outro qualquer na rede ele deve primeiro enviar sua mensagem para o servidor e o servidor, por sua vez, se encarrega de enviar a mensagem para o processo destino. Os processos, neste caso, seriam os clientes.

O início da comunicação acontece da seguinte forma: o servidor escolhe uma determinada porta de comunicação e fica aguardando conexões nesta porta. O cliente deve saber previamente qual a máquina servidora (*host*) e a porta que o servidor está aguardando conexões. Então o cliente solicita conexão em um *host*/porta. Se nenhum problema ocorrer, o servidor aceita a conexão gerando um *socket* em uma porta qualquer do lado servidor, criando assim um canal de comunicação entre o cliente e o servidor e depois armazena o *socket* específico de cada cliente em uma tabela de registros. Tipicamente o servidor tem duas funções: ficar em *loop* aguardando novas conexões e gerando *sockets* para atender as solicitações dos clientes (para cada conexão estabelecida com o cliente uma nova *thread* é criada e então o servidor volta a aguardar novas conexões) e também transmitir a mensagem de um cliente (processo) para outro usando uma tabela de registro de todos os clientes.

```
// sends a message to destination process
public void sendToPrcs(signal packet) throws IOException {
    try {
        sOut = (ObjectOutputStream) regClientesHT.get(packet.getPrcsDest());
        if(packet.getSignal()==null) {
            sOut.writeObject(null);
        } else {
            sOut.writeObject(packet);
        }
        sOut.flush();
    } catch (IOException e) {
        System.out.println("IOException: " + e);
    }
}
```

Figura 5.2: Método responsável por enviar sinal do servidor a processo-destino

Foi desenvolvido um sistema para que o servidor possa redirecionar o sinal de um processo ao processo-destino. Foi criada uma classe chamada ***signal*** para padronizar o sinal enviado entre os processos. Esta classe é composta de três atributos: processo destino, nome do sinal e lista de parâmetros do sinal. Quando um processo necessita enviar um sinal a outro processo, ele envia o sinal (objeto da classe ***signal*** contendo processo-destino, nome do sinal e lista de parâmetros) ao servidor. No servidor foi criado um método responsável por receber o sinal e enviá-lo para o processo-destino (Figura 5.2). O método recebe como entrada o sinal e, depois, analisa o atributo referente ao processo-destino para procurar tal processo na tabela de clientes registrados. A partir daí a tabela devolve o *socket* específico do processo-destino e assim o sinal é enviado para o mesmo.

5.4. Conversão de elementos da linguagem SDL

Além das técnicas de conversão apresentadas na seção 5.1 para o mapeamento de estruturas SDL em estruturas Java, algumas características da linguagem SDL tiveram que ser simuladas de alguma forma em Java por não possuírem um equivalente direto. Algumas dessas técnicas foram baseadas no trabalho de [23], tais como: construção de temporizadores, conversão de estados/transições e conversão de operações com *join*. Porém, o gerador desenvolvido em [23] não cobre todos os elementos SDL encontrados na modelagem da rede FlexRay deste trabalho. Sendo assim, tais elementos foram introduzidos ao conversor para possibilitar a cobertura integral dos elementos em SDL. Nesta seção são apresentadas algumas dessas novas funcionalidades introduzidas ao gerador. As principais funcionalidades introduzidas foram:

- Suporte a diversos níveis de blocos (*substructure*)

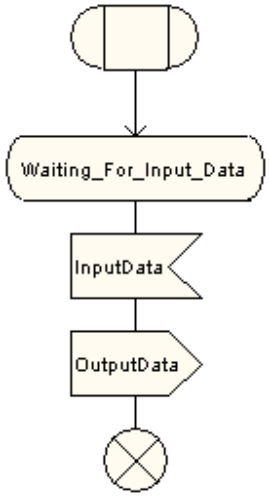
- Suporte a operações com *priority input*
- Suporte a operações com *save*
- Suporte a entrada de sinais em procedimentos
- Geração de código Java para dados do tipo *synonym*
- Geração de código Java para dados do tipo *syntype*
- Geração de código Java para dados do tipo *newtype*

De acordo com [22] sobre a hierarquia de um modelo SDL, um bloco pode conter processos ou, exclusivamente, outros blocos. A rede FlexRay foi modelada em SDL usando diversos níveis de blocos, sendo assim esta funcionalidade teve de ser incorporada ao gerador através da definição de *substructures*.

O suporte a entrada de sinais em procedimentos foi outro recurso habilitado neste gerador. Basicamente, esse recurso permite que os símbolos de entrada de sinal possam ser usados dentro de um procedimento e não apenas em processos. A rede FlexRay projetada faz extensivo uso desse recurso, logo o gerador de código Java deveria ser capaz de suportá-lo. A técnica proposta por este trabalho para simular a entrada de sinais em procedimentos consiste na implementação de um sistema exclusivo de recepção de sinais captados dentro do procedimento.

A Figura 5.3 mostra um exemplo da especificação em SDL referente à entrada de sinal em procedimento e como esse diagrama foi convertido para Java. Nesta figura o procedimento ***INPUTINPROCEDURE*** alcança o estado ***Waiting_For_Input_Data*** e ali permanece até a chegada do sinal ***InputData*** para depois enviar o sinal ***OutputData***. No código Java equivalente, ao entrar no procedimento ***INPUTINPROCEDURE*** (método Java definido dentro do processo correspondente) o estado do processo é alterado para o estado equivalente a ***Waiting_For_Input_Data***, então o programa entra no *while* (a variável ***HOLDPRCD*** recebeu o valor ***true*** no início do método) e pára no *if* responsável por aguardar pela chegada de algum sinal enviado àquele processo (a fila de sinais para procedimentos é a mesma do processo correspondente). Quando o sinal chega, o método ***commandSignal*** faz uma chamada ao método do sinal correspondente. Assim, o programa entra no método ***INPUTDATA***, verifica o estado atual do processo, envia o sinal ***OutputData*** e, por fim, atribui o valor ***false*** à variável ***HOLDPRCD***. Após isso o programa sai do método ***INPUTDATA***, retorna ao método ***INPUTINPROCEDURE*** e então o *while* é avaliado novamente. Como o valor de ***HOLDPRCD*** é ***false*** o programa sai do *while* e o método ***INPUTINPROCEDURE*** encerra sua execução.

Como mostrado por este exemplo, o código gerado possui exatamente a mesma funcionalidade encontrada na especificação em SDL.



CÓDIGO JAVA PARA A ENTRADA DE SINAL EM PROCEDIMENTO

```
public void INPUTINPROCEDURE (java.lang.Object senderParam___)
{
  ___sender = senderParam___;
  HOLDPRCD = true;
  try{
      ___changeState(___STATE_INPUTINPROCEDURE_WAITING_FOR_INPUT_DATA);
      while (HOLDPRCD)
      {
        if((data=(signal)sIn.readObject())!=null || !signalQueue.isEmpty())
        {
             if(!signalQueue.isEmpty())
             {
                  sig = (signal) getSignal();
                  commandSignal(sig);
             } else{
                  putSignal(data);
                  sig = (signal) getSignal();
                  commandSignal(sig);
             }
        }
      }
  } catch (IOException e) {
     System.out.println("Error in procedure INPUTINPROCEDURE");
     System.out.println(e.getMessage());
  } catch (ClassNotFoundException ex) {
     System.out.println("Error in procedure INPUTINPROCEDURE");
     ex.printStackTrace();
  } catch (Exception e) {
     System.out.println("Error in procedure INPUTINPROCEDURE");
     System.out.println(e.getMessage());
  }
```

```
}

public void INPUTDATA (java.lang.Object senderParam___)
{
 try{
     if ((___STATE ==___STATE_INPUTINPROCEDURE_WAITING_FOR_INPUT_DATA))
     {
       ___sender = senderParam___;
       JOptionPane.showMessageDialog(null,"OUTPUTDATA");
       HOLDPRCD = false;
      } else{}
 } catch (Exception e) {
    System.out.println(e.getMessage());
 }
}
```

Figura 5.3: Simulação da funcionalidade de entrada de sinal em procedimento

Dois novos elementos SDL que também foram habilitados no gerador foram o *priority input* e o *save*. O símbolo *priority input* é usado quando um sinal urgente precisa ser consumido antes dos outros sinais na fila, ao passo que o símbolo *save* é usado para salvar um sinal cujo símbolo de entrada não tenha sido especificado naquele estado atual, mas sim no estado seguinte. Nesse último caso, quando um sinal é salvo, ele permanece na mesma posição na fila de entrada e então o próximo sinal na fila é considerado [22].

A implementação em Java das construções *priority input* e *save* foi feita utilizando um conjunto de métodos e vetores inseridos dentro da classe gerada do processo correspondente. A criação desses métodos em Java representa a mesma funcionalidade das construções em SDL.

```
public void putSignal(signal sig) {
   prioritysig=false;
   if (saveHT.containsKey(sig.getSignal())) {
      state = (Integer) saveHT.get(sig.getSignal());
      if (___STATE == state) saveSignal(sig);
   }
   for (i=0; i<PRIORITYLIST.length; i++) {
      if (sig.getSignal().equalsIgnoreCase(PRIORITYLIST[i])) {
         prioritysig=true;
         break;
      }
   }
   if(prioritysig) signalQueue.add(0, sig);
   else signalQueue.add(sig);
}
```

Figura 5.4: Método responsável pela inserção de cada sinal na fila de sinais do processo

A Figura 5.4 ilustra a implementação do método ***putSignal***. O método ***putSignal*** é responsável pela inserção de cada sinal à fila de sinais e também verifica se o sinal tem de ser salvo ou se é um sinal com prioridade de entrada ou nenhum deles. Esse método utiliza uma estrutura de dados (classe do tipo *hashtable*), chamada ***SaveHT***, responsável por armazenar todos os sinais a serem salvos juntamente com a indicação do estado onde devem ser salvos. Assim, quando um sinal chega, ele é procurado no ***SaveHT*** e, se encontrado, seu estado é comparado ao estado atual do processo. Caso a verificação do estado seja positiva o método ***saveSignal*** é chamado. Já os sinais de entrada prioritária são armazenados em um vetor chamado ***PRIORITYLIST***. Quando um sinal chega, ele é procurado neste vetor e, se encontrado, é inserido na primeira posição na fila de entrada. A variável ***signalQueue*** é um vetor que representa a fila de sinais que chegam ao processo.

```
public void saveSignal (String signal){
  saved.add(signal);
}
```

Figura 5.5: Método responsável por adicionar um sinal a uma fila exclusiva de sinais salvos

A Figura 5.5 ilustra a implementação do método ***saveSignal***. O método Java ***saveSignal*** corresponde a construção *save* em SDL. Nesse método o sinal é salvo em um vetor para processamento posterior. A variável ***saved*** é um vetor que armazena os sinais a serem salvos.

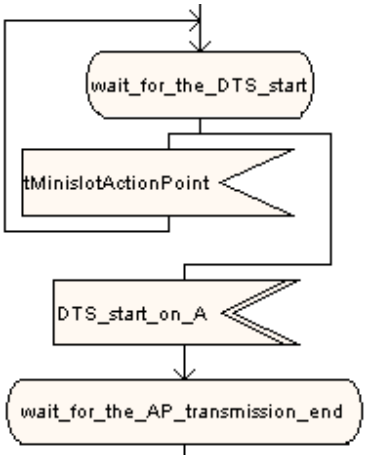
```
public Object getSignal() {
   Object ret, tmp;
   If (saved.contains(signalQueue.firstElement())){
      if (signalQueue.size()==1){
            return "";
      }
      tmp = signalQueue.remove(0);
      ret = signalQueue.remove(0);
      signalQueue.add(0,tmp);
      saved.clear(tmp);
      return ret;
   } else{
      ret = signalQueue.remove(0);
      return ret;
   }
}
```

Figura 5.6: Método responsável por consumir um sinal na fila de sinais do processo

A Figura 5.6 ilustra a implementação do método ***getSignal***. O método ***getSignal*** é responsável pelo gerenciamento da fila dos sinais de entrada do processo. Esse método sempre retorna o primeiro sinal armazenado na fila (***signalQueue***). A única exceção ocorre quando esse sinal foi salvo (armazenado no vetor ***saved***). Nesse caso o sinal seguinte na fila é retornado, o sinal salvo é removido do vetor ***saved*** e recolocado na primeira posição da fila.

O último método criado foi o ***initialize***. Ele foi projetado para ser executado no início de cada objeto Java gerado (processo SDL). Esse método é responsável por registrar a ocorrência de sinais modelados com *priority input* ou com *save*. Se no processo SDL tiver sido especificado algum sinal em alguma construção *priority input*, então no método ***initialize*** do objeto Java gerado será adicionado o sinal correspondente no vetor ***PRIORITYLIST***. No caso de algum sinal tiver sido especificado usando a construção *save*, então no método ***initialize*** será adicionado o sinal correspondente no vetor ***SaveHT***.

A Figura 5.7 mostra a simulação da funcionalidade de *priority input*. Neste exemplo o sinal ***DTS_start_on_A*** tem prioridade sobre o sinal ***tMinislotActionPoint***. Assim, ao entrar no estado ***wait_for_the_DTS_start*** a fila de sinais é verificada, caso o sinal ***DTS_start_on_A*** esteja na fila ele será consumido primeiro. No código Java equivalente o sinal ***DTS_start_on_A*** é armazenado no vetor ***PRIORITYLIST*** dentro do método ***initialize*** e será verificado no método ***putSignal***.



CÓDIGO JAVA PARA ENTRADA DE SINAL PRIORITÁRIO *DTS_start_on_A*

```
public void initialize () {
  PRIORITYLIST = new String[1];
  PRIORITYLIST[0] = “DTS_START_ON_A”;
}
```

Figura 5.7: Simulação da funcionalidade de PRIORITY INPUT em Java

A Figura 5.8 mostra a simulação da funcionalidade de *save*. Neste exemplo o sinal ***tSlotBoundary*** é especificado em uma construção *save*, visto que seu símbolo de entrada não foi especificado em nenhum estado dentro daquele procedimento, mas sim no estado seguinte do processo corrente. No código Java equivalente o sinal ***tSlotBoundary*** é armazenado no vetor ***SaveHT*** dentro do método ***initialize*** para, a posteriori, também ser verificado no método ***putSignal***.

CÓDIGO JAVA PARA CONSTRUÇÃO SAVE
tSlotBoundary

```
public void initialize () {
  saveHT.put(“TSLOTBOUNDARY”, new Integer(23));
}
```

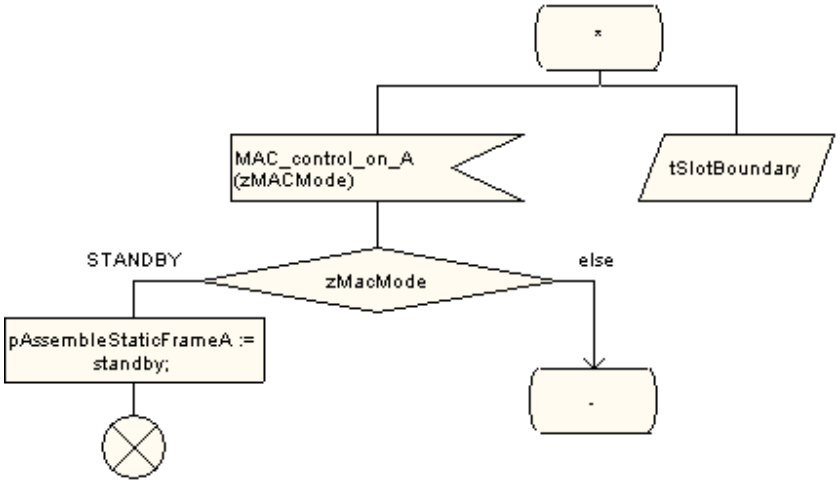


Figura 5.8: Simulação da funcionalidade de SAVE em Java

Outras funcionalidades habilitadas no gerador foram as estruturas de dados *syntype*, *synonym* e *newtype*.

A estrutura *syntype* é usada tanto para definir o tipo de uma variável como também para definir uma variável com intervalo de valores. O gerador desenvolvido em [23] suporta *syntype*

para definição do tipo da variável, mas não para definição de uma variável com intervalo de valores. Sendo assim, a estrutura *syntype* foi implementada em Java como sendo uma classe representando o tipo de dado correspondente. A Tabela 5.2 descreve como um intervalo de valores é definido em SDL usando a construção *syntype* e como o mesmo é convertido para Java usando a classe correspondente.

Tabela 5.2: Conversão de tipo syntype de SDL para Java

Descrição do tipo *syntype* para definição de faixas de valores de uma constante em SDL	**Definição de faixas de valores para uma constante usando classe em Java**

syntype T_FrameID = Integer
constants 0 : 20
endsyntype;
	public final class T_FrameID implements _Constants {
	public T_FrameID (int value) throws Exception
	{
	if ((value >= lowerBound) &&
	(value <= upperBound)){
	this.value=value;
	}else{
	throw new Exception("Value out of
	predefined bounds!")
	}
	}
	public int value(){
	return value;
	}
	public void setValue(int val){
	value = val;
	}
	private int value;
	private static final int upperBound = 20;
	private static final int lowerBound = 0;
	}

As estruturas *synonym* e *newtype* sofreram modificações em sua forma de conversão para o Java devido a um problema semelhante. A ferramenta desenvolvida em [23] não gera código Java para estas funcionalidades, mas sim gera sua representação em IDL. Logo, o gerador deste trabalho foi projetado para gerar código puramente em Java para essas estruturas.

A estrutura de dados *synonym* é usada para definir constantes. O artifício utilizado para converter essa estrutura para código Java foi usar atributos de *interface* Java. A *interface* ***_Constants*** é responsável por armazenar todas as constantes definidas no projeto. A Tabela 5.3 mostra a conversão do termo *synonym* em SDL por *public static final* em Java.

Tabela 5.3: Conversão de constantes de SDL para Java

Definição de constantes através do uso da construção *synonym* em SDL	**Definição de constantes através do uso de atributo de interface em Java**

synonym gdNIT Integer = 50;
synonym gdMinislot Real = 40;
synonym gdMacrotick Duration = 40;
	public interface _Constants {
	  public static final int GDNIT = 50;
	  public static final double GDMINISLOT = 40;
	  public static final long GDMACROTICK = 40;
	}

A estrutura *newtype* tem por finalidade criar novos tipos de dados em SDL. A fim de converter essa estrutura para código Java foi necessário representá-la como uma classe Java correspondente. Dessa forma a classe gerada é integralmente codificada em Java e totalmente funcional. A Tabela 5.4 exemplifica a conversão da estrutura *newtype* para classe Java através do tipo ***T_TransmitFrame***.

Tabela 5.4: Conversão da estrutura newtype para classe Java

Definição do tipo *T_TransmitFrame* através da estrutura *newtype* em SDL

```
newtype T_TransmitFrame struct
   Header T_Header;
   Payload T_Payload;
endnewtype;
```

Definição do tipo *T_TransmitFrame* através da definição de classe em Java

```
package FLEXRAY.DECLARATION;

public final class T_TRANSMITFRAME implements _Constants, java.io.Serializable
{
   public T_PAYLOAD PAYLOAD;
   public T_HEADER HEADER;

   public T_TRANSMITFRAME() {}

   public T_TRANSMITFRAME (T_PAYLOAD _PAYLOAD, T_HEADER _HEADER)
   {
      PAYLOAD = _PAYLOAD;
      HEADER = _HEADER;
   }

   public boolean equals(T_TRANSMITFRAME param)
   {
      return ((PAYLOAD.equals(param.PAYLOAD)) && (HEADER.equals(param.HEADER)));
   }

   public static T_TRANSMITFRAME ___Random()
```

```
    {
        T_TRANSMITFRAME ___ret = new T_TRANSMITFRAME();
        ___ret.PAYLOAD = ((T_PAYLOAD.___Random()));
        ___ret.HEADER = ((T_HEADER.___Random()));
        return ___ret;
    }
}
```

5.5. Interface gráfica do programa gerado em Java

Um sistema SDL se comunica com uma aplicação externa ou com o usuário através do ambiente (*environment*). Como os sinais trocados com o ambiente representam a entrada e saída de dados no sistema foi desenvolvida uma interface gráfica para o programa gerado em Java que desempenha função equivalente.

O desenvolvimento da interface gráfica foi baseada na implementação feita em [23] por se tratar de uma forma simples e eficiente de interação com o usuário.

Quando um processo SDL se encontra em um determinado estado, um conjunto de sinais pode ser recebido do usuário. No sistema equivalente em Java, dependendo do estado em que se encontre o objeto, os sinais são disponibilizados ao usuário através de uma lista de sinais que podem ser transmitidos ao sistema naquele momento. A Figura 5.9 ilustra a escolha do sinal a ser enviado ao sistema.

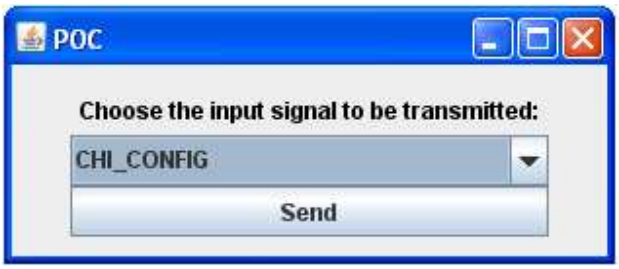


Figura 5.9: Envio de sinal ao sistema

Caso o sinal a ser transmitido necessite de configuração de parâmetros, uma nova janela aparecerá para que o usuário possa preencher os parâmetros apropriadamente. Desta forma, pode-se então estabelecer uma comunicação entre usuário e sistema de forma similar ao modelo especificado em SDL. A Figura 5.10 ilustra a janela que solicita ao usuário o preenchimento dos parâmetros.

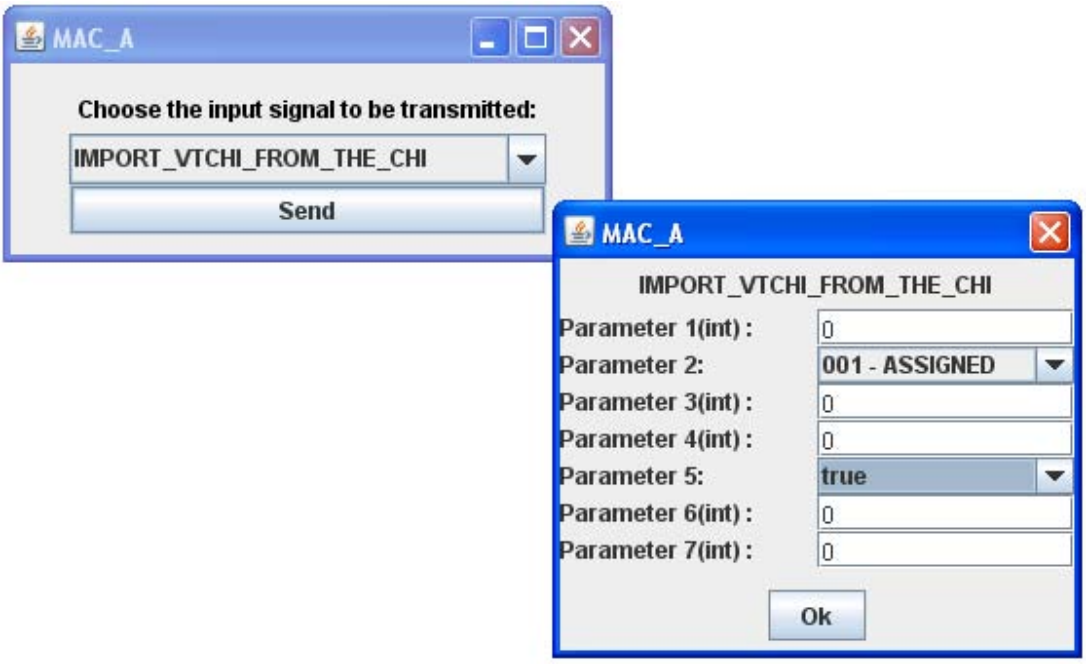


Figura 5.10:Solicitação de preenchimento dos parâmetros de determinado sinal

Para representar os sinais enviados do sistema para o ambiente, foi utilizado um recurso gráfico do Java de exibição de mensagem ao usuário: a caixa de mensagem. Tais caixas de mensagem exibem o sinal recebido e seus parâmetros. A Figura 5.11 ilustra a caixa de mensagem contendo um sinal recebido do sistema.

Figura 5.11:Sinal recebido do sistema

A vantagem de ter uma interface gráfica para a rede codificada em Java é a possibilidade de simulação da troca de informações entre o sistema e o ambiente. Dessa forma pode-se verificar se o sistema responderá corretamente caso uma aplicação real seja conectada à rede FlexRay implementada, já que o usuário fará o papel desempenhado pela aplicação.

Capítulo 6

Conclusão

Recentemente tem havido um aumento significativo na quantidade de eletrônica introduzida no carro, e é esperado que esta tendência continue enquanto as montadoras introduzirem avanços adicionais em segurança, confiabilidade e conforto. A introdução de sistemas avançados de controle está começando a demandar exigências na tecnologia de uma comunicação não encontrada atualmente nos protocolos de comunicação existentes. Em virtude disso o protocolo FlexRay surge como o sistema de comunicação mais adequado para sistemas automotivos futuros de alta velocidade.

Como o protocolo FlexRay tem um futuro muito promissor e com o objetivo de ajudar no desenvolvimento e pesquisa deste sistema, este trabalho propôs uma implementação de uma rede FlexRay através de um modelo SDL funcional e executável. Junto com a modelagem da rede FlexRay em SDL também foi desenvolvido um compilador de SDL para Java de forma a facilitar a implementação de uma aplicação específica.

Para criar uma rede FlexRay completa e funcional sua estrutura teve que ser projetada, já que estes detalhes não estão descritos na especificação do protocolo. Dessa forma, a arquitetura do sistema, a topologia da rede, a integração entre os principais mecanismos dentro do controlador de comunicação e a interface que conecta o controlador de comunicação ao canal de comunicação foram todos criados e descritos usando a linguagem SDL.

Além da descrição completa da estrutura da rede proposta no capítulo 3, algumas adaptações também foram feitas em relação à especificação do FlexRay a fim de gerar um sistema executável usando SDL. O modelo SDL gerado confere uma formalização comportamental bastante interessante ao sistema.

O uso de uma ferramenta *case* para o desenvolvimento profissional em SDL, como o *SDL TAU Suite* (SDT) [12], traz grandes vantagens para realização das especificações. A utilização de uma representação gráfica da linguagem SDL (SDL-GR) facilita bastante o processo de

especificação. No capítulo 4, a realização do processo de validação das especificações desenvolvidas permite a detecção e correção de erros na especificação. Além disso, a análise dos resultados de simulações através de diagramas MSC gerados pelo simulador é de compreensão bastante simples e permite a detecção de erros de lógica no sistema. Tais diagramas podem ser utilizados na criação de pacotes de testes para implementação de qualquer aplicação usando o FlexRay. Estes pacotes de testes podem ser produzidos por ferramentas específicas baseadas na notação TTCN (*Tree and Tabular Combined Notation*) [21], parte integrante do SDT, com o intuito de automatizar o processo de testes.

A conversão automática das especificações desenvolvidas em SDL para código Java através da ferramenta desenvolvida (capítulo 5), permite a execução do sistema modelado e oferece ao projetista uma interação com o sistema. Foram propostas algumas soluções para a conversão de características da linguagem SDL que não possuem equivalente direto em Java e que não foram cobertas pelo gerador desenvolvido em [23]. O suporte a entrada de sinais em procedimentos e a implementação das construções *priority input* e *save* foram algumas das soluções desenvolvidas para a ferramenta proposta neste trabalho.

Apesar de neste trabalho a ferramenta desenvolvida ter sido utilizada para a geração de código para uma modelagem de sistema FlexRay, sua utilização é mais extensa, podendo converter para Java qualquer especificação em SDL, desde que siga a estrutura de rede proposta no capítulo 3 e que aplique as regras de mapeamento propostas no capítulo 5.

Baseado na metodologia proposta para implementação de uma rede FlexRay através de um modelo SDL e na construção da ferramenta geradora de código Java desenvolvida pode-se propor a realização de alguns trabalhos futuros:

- Especificação em SDL dos processos de sincronismo do *clock* (CSS, CSP e MTG) como parte do controlador de comunicação. Tais processos foram considerados fora do escopo do sistema proposto nesse trabalho, mas sua interação com o sistema manteve-se através da comunicação pelo ambiente;
- Especificação em SDL do processo CODEC. Neste trabalho, foi idealizado um esquema de adaptação do comportamento do processo CODEC para embutí-lo no controlador de barramento. Dessa forma eliminou-se a complexidade do processo sem perder sua utilidade. Seria interessante, porém, a especificação completa deste processo e sua integração com todos os processos do sistema FlexRay;

- Projetar novas topologias de redes FlexRay. Neste trabalho foi considerada uma rede de um único canal configurado na topologia de barramento interconectando três nós. No entanto, de acordo com a especificação técnica, há várias maneiras de se projetar uma rede FlexRay. Ela pode ser configurada como uma rede do tipo barramento com um ou dois canais, uma rede estrela com um ou dois canais ou várias outras combinações híbridas de topologia de barramento e estrela.
- Geração de pacotes de testes através da notação TTCN [21] utilizando os diagramas MSCs gerados na simulação da rede proposta nesse trabalho (capítulo 4). Tal procedimento visaria automatizar o processo de testes de implementações para qualquer aplicação usando o sistema FlexRay.

Referências Bibliográficas

[1] FlexRay Consortium. FlexRay. Disponível em: <http://www.flexray.com> Acesso: Janeiro 2008.

[2] E. Armengaud et al. “A monitoring concept for an automotive distributed network - the FlexRay example”. In: 7th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2004), 2004.

[3] G. Leen, D. Hefferman, “In-vehicle networks, expanding automotive electronic systems”. In: IEEE Transaction on Computers, Jan. 2002, pp. 88-93.

[4] J. Berwanger et al. “FlexRay – the communication system for advanced automotive control systems”. In: SAE 2001 World Congress, Society of Automotive Engineers, Detroit, MI, Apr. 2001.

[5] FlexRay Communications System - Protocol Specification Version 2.1 Revision A, FlexRay Consortium, Dezembro 2005.

[6] T. Nolte, H. Hansson, L. Bello, “Implementing next generation automotive communications”. In: Proceedings of the 1st Embedded Real-Time Systems Implementation Workshop (ERTSI'04) in conjunction with the 25th IEEE International Real-Time Systems Symposium (RTSS'04), Lisbon, Portugal, Dec. 2004.

[7] A. Albert. “Comparison of event-triggered and time-triggered concepts with regard to distributed control systems”. In: Embedded World, Nurnberg, Germany, 2004.

[8] R. Braek, “SDL basics”. In: Computer Networks and ISDN Systems, Jun. 1996, pp. 1585-1602.

[9] L. Doldi, Validation of Communications Systems with SDL. Chichester, Hoboken (NJ), Wiley, 2003.

[10] Society of Automotive Engineers. Disponível em: <http://www.sae.org>. Acesso: Abril 2009.

[11] SDL Forum Society. Disponível em: <http://www.sdl-forum.org>. Acesso: Abril 2009.

[12] Telelogic AB. Telelogic TAU 4.2 SDL Suite Getting Started: Technical report, Telelogic AB Sweden, Sep. 2001.

[13] D. Rezende, W. Borelli. “Proposta de uma Rede de Comunicação Automotiva Baseada no FlexRay”. In: XXV Simpósio Brasileiro de Telecomunicações (SBrT’07), Recife, PE, Setembro 2007.

[14] ITU-T, Genebra, Suíça. Specification and Description Language, 1993. (CCITT Recommendation Z.100).

[15] Telelogic AB, Suécia. Telelogic Tau 4.2 SDL Suite Methodology Guidelines, Setembro 2001.

[16] Metamata Inc. Java Compiler Compiler (JavaCC) - The Java Parser Generator, 2000.

[17] ITU-T, Genebra, Suíça. Message Sequence Chart (MSC), 1993. (CCITT Recommendation Z.120).

[18] D. Paret, Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire. Chichester, Wiley, 2007.

[19] FlexRay Communications System - Electrical Physical Layer Specification, v2.1 Revision A, FlexRay Consortium, Dezembro 2005.

[20] G. Holzmann, Design and Validation of Computer Protocols. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[21] CCITT Recommendation X.292, OSI conformance testing methodology and framework for protocol Recommendations for CCITT applications – The Tree and Tabular Combined Notation (TTCN), 1992.

[22] L. Doldi, SDL Illustrated: Visually design executable models. Laurent Doldi, 2001.

[23] R. Guimarães and W. Borelli, “An Automatic Java Code Generation Tool for Telecom Distributed Systems”. In: 10th International Conference on Software Telecommunication (SOFTCOM’2002).

[24] M. Litterick and M. Brenner, “Utilizing Vera Functional Coverage in the Verification of a Protocol Engine for the FlexRay Automotive Communication System”, SNUG Europe, 2005.

[25] G. Csopaki, G. Horváth, G. Kovács, “Communication Protocol Implementation in Java”. In: Interactive Distributed Multimedia Systems and Telecommunication Services, Jan. 2000, pp. 1201-1205.

[26] C. Horstmann, G. Cornell. CoreJava 2 - Volume I - Fundamentos. MAKRON Books, São Paulo, 2001.

[27] M. Delamaro, Como Construir um Compilador Utilizando Ferramentas Java. Novatec, 2004.

[28] A. Guimarães, Eletrônica Embarcada Automotiva. Érica, 2007.

[29] A. Oliveira, F. Andrade, Sistemas Embarcados – Hardware e Firmware na Prática. Érica, 2006.

[30] J. Rushby, "Bus Architectures For Safety-Critical Embedded Systems". In: First Workshop on Embedded Software (EMSOFT'2001), Out. 2001, pp. 306–323.

[31] E. Sherratt, C. Loftus. “Designing distributed services with SDL”. IEEE Concurrency, 08(01):59–66, Janeiro-Março 2000.

[32] C. Kavadias, B. Perrin, V. Kollias, M. Loupis, "Enhanced SDL Subset for the Design and Implementation of Java-Enabled Embedded Signalling Systems". SDL Forum, 2003, pp. 137-149.

[33] Vector Informatik GmbH. DaVinci Network Designer FlexRay. Disponível em: <http://www.vector.com>. Acesso: Julho 2009.

[34] TTTech Computertechnik AG. TTXPlan. Disponível em: <http://www.tttech-automotive.com>. Acesso: Julho 2009.

[35] LIN - Protocol, Development Tools, and Software Interfaces for Local Interconnect Networks in Vehicles, 9th International Conference on Electronic Systems for Vehicles, Baden-Baden, October 2000.

[36] Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. International Standards Organisation (ISO). ISO Standard-11898, Nov 1993.

[37] MOST Specifcation Framework Rev 1.1, MOST Cooperation. Disponível em: <http://www.mostnet.de>. Acesso: Julho 2009.

[38] Time-Triggered Protocol TTP/C, High-Level Specification Document, Specification edition 1.0.0 of 4-July-2002, TTTech Computertechnik AG.

Apêndice A

A linguagem SDL e a ferramenta SDL TAU Suite

A linguagem SDL (*Specification Description Language*) [14] foi publicada primeiramente como uma recomendação do CCITT (*Comité Consultatif International Téléphonique*), no começo dos anos 70. Atualmente o órgão CCITT passou a ser conhecido como ITU-T (*International Telecommunications Union - Telecommunication*).

O SDL é uma linguagem para a especificação e descrição de protocolos, sistemas de telecomunicações e sistemas de tempo real. O SDL se concentra basicamente na especificação dos aspectos comportamentais do sistema, envolvendo dados quando necessário. A linguagem é ao mesmo tempo textual e gráfica, ou seja, os sistemas especificados em SDL podem fazer uso de componentes gráficos da linguagem, os quais possuem equivalentes textuais.

O SDL possui algumas características que o diferencia de outras linguagens. É uma linguagem padrão, formal, orientada a objetos, portável, escalável, um padrão aberto e permite reuso.

A.1. Estrutura do sistema

Sistemas SDL podem ser estruturados de várias formas. Um sistema (*system*) consiste de blocos (*block*) conectados por canais (*channel*), cada bloco pode conter uma subestrutura de blocos (*substructure*) ou pode conter um conjunto de processos (*process*) conectados por rotas de sinais (*signal routes*).

As especificações SDL podem ser modularizadas por meio de pacotes (*packages*). Um pacote é uma coleção de definições de tipos. Pacotes também podem ser utilizados na definição de novos pacotes ou na definição de sistemas.

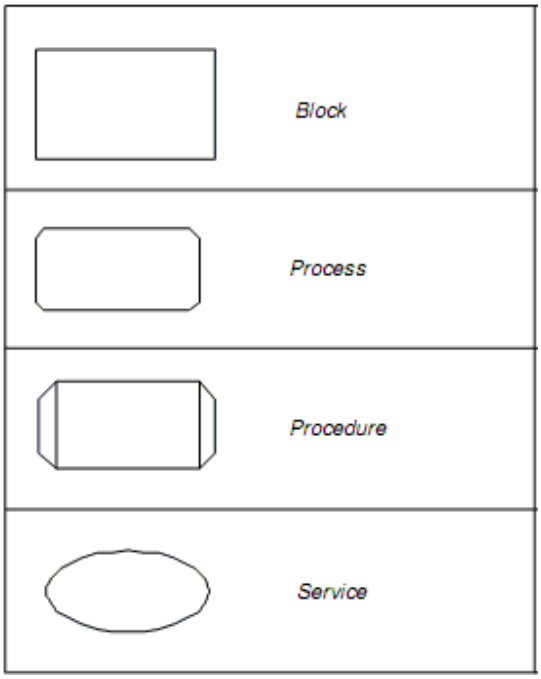


Figura A.1: Estruturas básicas do SDL

A Figura A.1 ilustra algumas estruturas básicas do SDL. Dentre elas pode-se destacar:

System: É a entidade mais externa, que representa o sistema como um todo. É composto por diversos blocos que se comunicam entre si e com o ambiente através de canais de sinais. Os sinais e tipos de dados utilizados pelo sistema e que devem ser conhecidos por diversos blocos devem ser declarados neste nível.

Block: É a entidade que agrupa um ou mais processos que se comunicam entre si e com os canais de comunicação do sistema (podendo se comunicar com outros blocos ou com o ambiente). Os sinais e tipos de dados utilizados pelo bloco e que ainda não foram declarados no nível do sistema devem ser declarados neste nível.

Process: É a estrutura responsável pela especificação comportamental do sistema, através da descrição de uma máquina de estados finita com dados, que é independente das demais. O conjunto de processos que são executados em paralelo é que definem o comportamento do sistema.

Procedure: Da mesma forma que nas linguagens de programação, agrupam funcionalidades que se repetem em diversas partes do sistema ou então agrupam funcionalidades de forma a simplificar a análise do sistema. Um procedimento pode ou não receber parâmetros para o seu

processamento e, da mesma forma, pode ou não retornar um resultado. Além disso, todas as variáveis declaradas no processo são visíveis pelos procedimentos contidos no mesmo.

Service: Usado para definir parte do comportamento de um processo. Em algumas situações o *service* pode reduzir a complexidade e aumentar o entendimento da especificação de um processo.

A.2. Troca de sinais

Os sinais são transportados de um bloco para outro ou para o ambiente externo (*environment*) através de canais de comunicação (representados por *c1*, *c2* e *c3* na Figura A.2). Cada um destes canais pode ser uni ou bidirecional e possui listas que indicam quais sinais são trocados em cada direção do canal. Para a troca de sinais entre processos faz-se uso de rotas de sinais (representados por *r1* e *r2* na Figura A.2), os quais possuem uma estrutura similar aos canais de comunicação. Os processos também fazem uso de rotas de sinais para se comunicar com o nível mais externo, conectando uma das extremidades da rota de sinais a um canal de comunicação do nível de sistema. Neste caso, os sinais transportados pelas rotas de sinais devem ser os mesmos transportados pelo canal de comunicação.

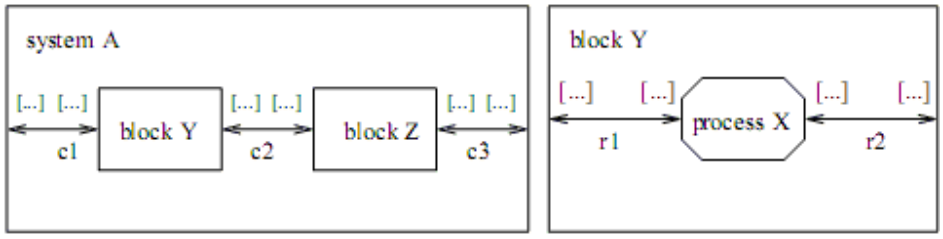
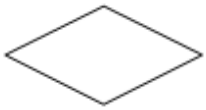
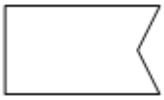
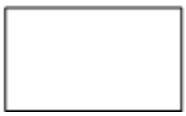
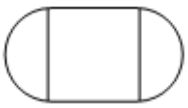


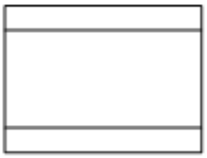
Figura A.2: Representação da troca de sinais em SDL

A.3. Principais elementos para a descrição de processos e procedimentos

As especificações realizadas dentro do processo e procedimento, por exemplo, na descrição de uma máquina de estado, fazem uso dos elementos apresentados na Tabela A.1.

Tabela A.1: Descrição e representação de algumas estruturas SDL

Representação	Descrição
	*Start:* utilizado apenas uma vez no processo indicando o início de sua execução
	*Stop*: utilizado para indicar o término da execução de um processo
	*State*: utilizado para indicar o estado em que se encontra a máquina finita que representa o processo/procedimento. Uma mudança de estado só ocorre quando se recebe um sinal. O estado pode ter um nome, ou pode ser declarado o estado '*' que se refere a todos os estados existentes, indicando que as ações a serem tomadas são comuns a todos eles. Além disso, após a recepção de um sinal (e execução das ações relacionadas), a execução pode encontrar o estado '-' que indica que o estado não se alterou devido às ações.
	*Decision*: utilizado para uma tomada de decisão. Consiste de uma questão e diversas possíveis respostas e, dependendo da resposta, a execução toma um caminho distinto.
	*Text*: utilizado para a declaração de variáveis, sinal e tipos de dados.
	*Input*: utilizado para indicar um sinal de entrada. Deve vir sempre após um estado indicando uma transição devido ao recebimento deste sinal.
	*Output*: utilizado para indicar um sinal de saída. Através desta estrutura enviam-se sinais e dados para outros processos. Para cada output deve haver um input associado no processo de destino. Para indicar o processo destino pode-se utilizar o endereçamento implícito - omitindo-se o endereço do destino - ou explícito - indicando-se o endereço destino através da opção TO <dest> ou indicando a rota do sinal a seguir através da opção VIA <rota>.
	*Task*: utilizado para definir uma tarefa ou um conjunto de tarefas (separados por ´;´). As tarefas podem ser substituídas por uma string com descrição textual das mesmas.
	*Return*: utilizado para indicar o término da execução de um procedimento. Indica um valor de retorno do procedimento, quando for o caso.
	*Start Procedure*: utilizado apenas uma vez no procedimento indicando o início de sua execução.

	*Create*: utilizado para criar uma instância de um processo, desde que esteja no mesmo bloco do processo criador.
	*Procedure Call*: utilizado para indicar uma chamada a um procedimento.
	*Comment:* utilizado para a inserção de comentários na especificação.

A.4. Package

Um dos conceitos mais importantes da linguagem SDL é o conceito de *package*. O *package* permite que uma definição de tipo seja usada por vários sistemas diferentes, similar ao objetivo da biblioteca de classes em C++.

O *package* permite a criação de uma biblioteca que pode ser utilizada em um sistema. No *package* podem ser definidos tipos de dados, listas de sinais, *System Types*, *Block Types*, *Process Types*, dentre outros, podendo ser utilizado integralmente ou parcialmente pelo sistema que o inclua através da cláusula *use*.

Além de poder ser incluído em um sistema, um *package* pode ser incluído em outro *package*, criando assim uma hierarquia de *packages*.

A.5. SDL TAU *Suite*

O SDL TAU *Suite* [12][15] é uma ferramenta *CASE* (*Computer Aided Design Software Engineering*), distribuída pela *Telelogic*, que utiliza a linguagem SDL para a especificação, validação e simulação dos sistemas. Dentre os componentes da versão utilizada, o SDL TAU *Suite* 4.4, pode-se destacar os seguintes:

Organizador: utilizado para a organização geral das especificações. Fornece uma visão de todos os diagramas e documentos que compõem o sistema, os quais podem ser livremente agrupados em capítulos e módulos de forma a facilitar a organização.

Editor SDL: possibilita a edição comportamental do sistema através do uso da representação gráfica da linguagem SDL.

Visualizador de Tipos: permite a visualização do impacto dos mecanismos de herança e especialização no sistema.

Analisador: realiza a análise sintática e semântica da especificação desenvolvida verificando a existência de erros e gerando código para os processos de simulação e validação.

Simulador: permite a simulação das características do sistema, tendo como saída um diagrama de troca de mensagens (MSC - *Message Sequence Chart*) que mostra o comportamento do sistema ao longo do tempo.

Validador: permite a validação do sistema, ou seja, percorre todos os possíveis estados em que o sistema pode se encontrar auxiliando assim na detecção de erros, como por exemplo o não tratamento de algum sinal em determinados estados.

Editor MSC: utilizado pelo simulador para a geração dos diagramas de troca de mensagens.

Visualizador de área coberta: utilizado pelo simulador e pelo validador para exibir uma árvore com todos os estados do sistema destacando aqueles alcançados pelo processo em questão (simulação ou validação).

Geradores de código: possibilitam a geração de código C, Cmicro e CHILL a partir das especificações em SDL.

A presença dos componentes Simulador e Validador é importante para a realização de testes da especificação. Através do uso destes componentes podem-se encontrar erros na especificação através da simulação de casos críticos, bem como através da análise dos estados alcançados pelo Validador com o uso do visualizador de área coberta (*Coverage Viewer*).

Além disso, o SDL TAU *Suite* permite a integração do desenvolvimento em SDL com outras linguagens como o ASN.1 (*Abstract Syntax Notation One*) e UML (*Unifing Modelling Language*), garantindo assim uma maior flexibilidade no processo de desenvolvimento. Existe também a possibilidade da realização de testes de casos de uso através do uso da linguagem TTCN (*Tree and Tabular Combined Notation*).

A Figura A.3 mostra a relação entre estas linguagens.

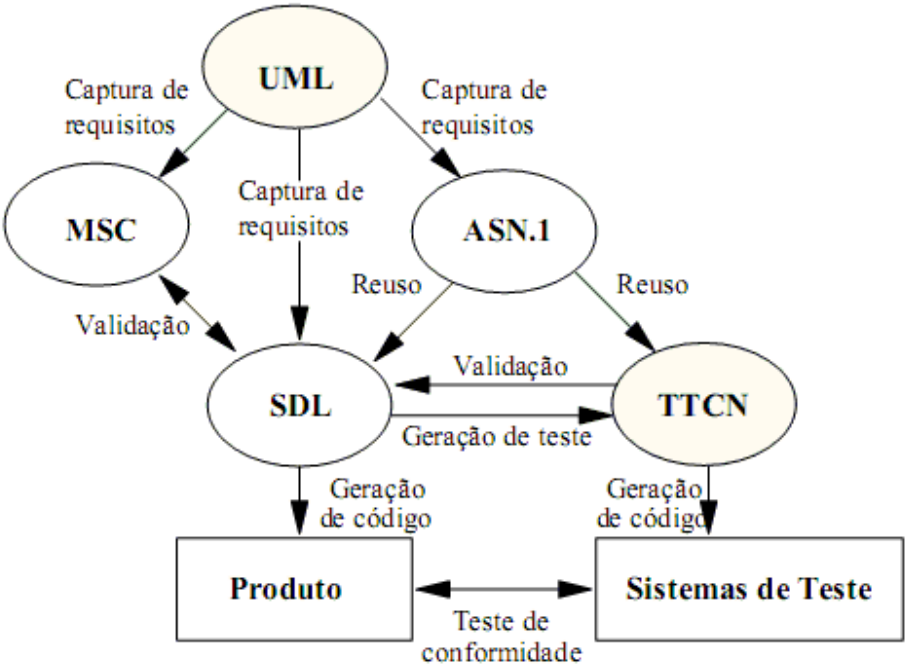


Figura A.3: Relação entre as linguagens que fazem interface com o SDL TAU *Suite*

A Figura A.4 mostra uma visão geral dos módulos componentes da ferramenta e suas relações.

O Organizador é responsável por gerenciar quase todos os componentes do pacote SDL TAU *Suite*. Através dele tem-se o acesso a todas as funcionalidades disponíveis. Tanto para a realização da simulação quanto da validação, as especificações passam primeiramente por uma análise (através do Analisador) que garante a exatidão sintática das especificações. Após esta análise, é gerado o código C, o qual é utilizado pelos módulos responsáveis pela realização das simulações e da validação. Ainda é disponibilizada uma biblioteca de aplicações que, quando integrada ao gerador de código C, permite a geração de uma implementação em C do sistema especificado e a execução do mesmo.

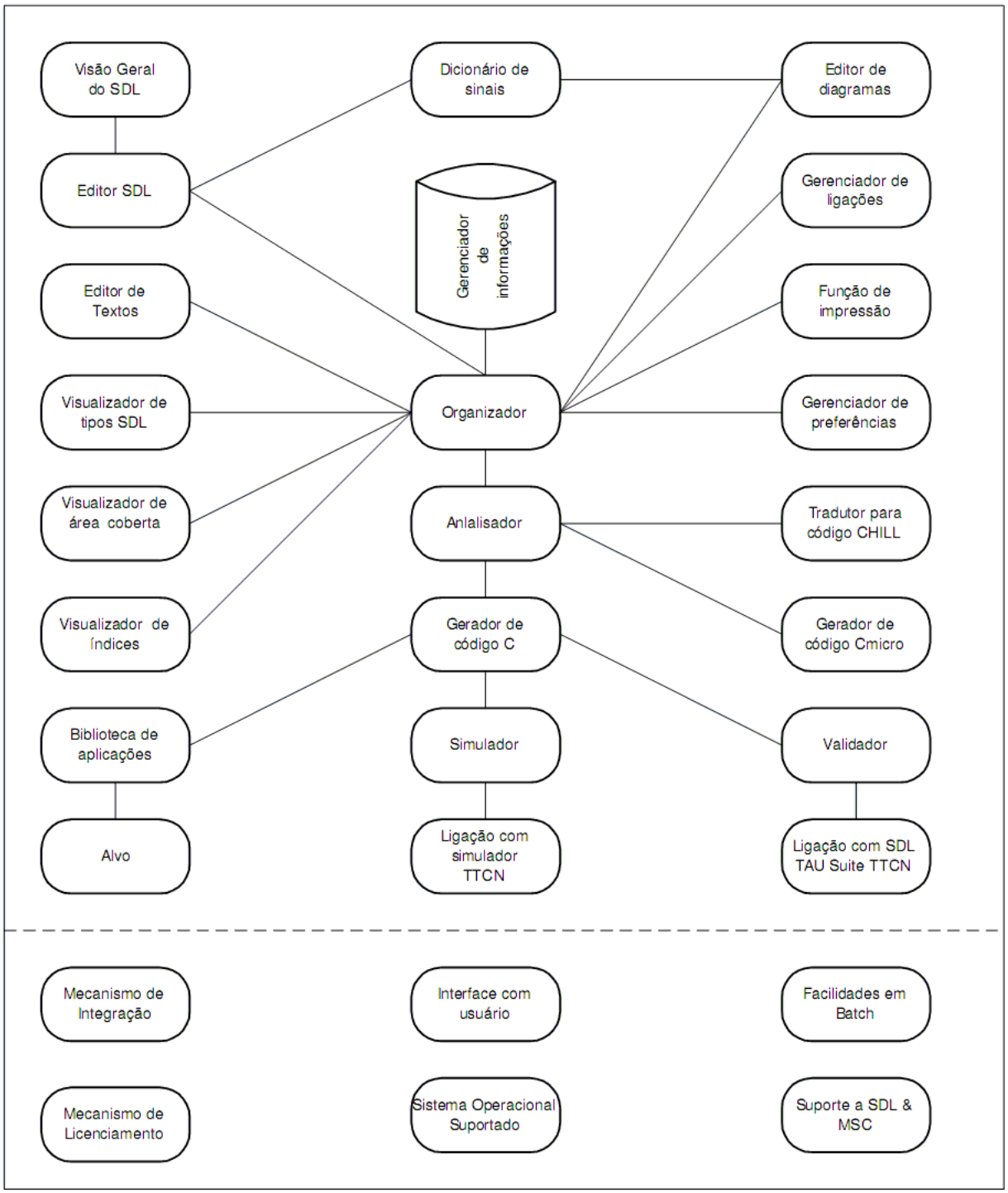


Figura A.4: Visão geral dos módulos que compõem o SDL TAU *Suite*

Apêndice B

Geração de código e JavaCC

Uma linguagem de programação (como o SDL) pode ser definida através da descrição de como seus programas se parecem (a sintaxe da linguagem) e o que eles significam (a semântica da linguagem).

Para a especificação da sintaxe, a notação mais utilizada é a chamada gramática de livre contexto ou BNF (*Back-Naur Form*).

Já a especificação da semântica de uma linguagem é mais complexa e, por isso, não possui uma notação adequada, devendo ser descrita informalmente, como por exemplo através de linguagem natural.

Além de especificar a sintaxe de uma linguagem, a notação BNF pode ser utilizada para auxiliar a tradução de programas de uma linguagem para a outra, ou seja, para organizar a estrutura de um compilador que realize esta tarefa.

B.1. O gerador de código Java

O gerador de código desenvolvido pode ser visto como um compilador que tem como saída um programa em Java. O processo de compilação possui diversas fases, sendo que a saída de uma fase é a entrada de outra.

Dentre as fases mais importantes do processo de compilação, podemos citar em ordem de ocorrência: analise léxica, analise sintática, análise semântica e geração de código.

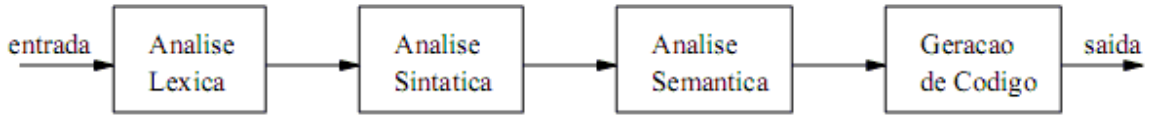


Figura B.1: Fases de um compilador

B.1.1. Análise Léxica

A análise léxica é a primeira fase de um compilador. Sua principal tarefa é a leitura dos caracteres de entrada e a geração de uma sequência de *tokens* (sequência de caracteres que possuem algum significado) que são utilizados na próxima fase (análise sintática).

O analisador léxico também pode ser responsável pela retirada de comentários presentes no programa de entrada (que são inúteis para no processo de compilação), bem como de caracteres de espaço, *tab* e *newline*.

Os caracteres da atribuição: ***tamanho := incremento * 10***, seriam agrupados nos seguintes *tokens*:

1. O identificador ***tamanho***
2. O símbolo de atribuição ***:=***
3. O identificador ***incremento***
4. O sinal de multiplicação *******
5. O número ***10***

Os espaços em branco são ignorados.

B.1.2. Análise Sintática

Após o processo de análise léxica, é necessário verificar se a sequência de *tokens* reconhecidos respeita a sintaxe esperada. Para isso, necessita-se representar essa sintaxe através do uso da notação BNF.

Uma chamada a um procedimento em SDL tem a seguinte forma:

call *identificador* ***(*** *lista de parametros* ***) to*** *expressão*

Isto significa que a chamada a um procedimento é a concatenação da palavra reservada *call*, de um abre-parênteses, de uma lista de parâmetros, de um fecha-parênteses, da palavra reservada *to* e de uma expressão. Utilizando as variáveis *id* para representar o identificador, *param* para representar a lista de parâmetros, *expr* para representar a expressão e *prcdcall* para representar a chamada ao procedimento em si, a estrutura acima pode ser representada por:

prcdcall → ***call*** *id* ***(*** *param* ***) to*** *expr*

onde a seta pode ser lida como “pode ser representada por”. Uma regra como esta é chamada produção e elementos como *id*, *param* e *expr* são chamados não-terminais, pois se refere a alguma outra produção. Além dos terminais, a notação BNF possui mais três tipos de

elementos: os terminais (*call*, *to*, abre e fecha parênteses), as produções (um não terminal, seguido de uma seta, seguido de uma sequência de terminais e não-terminais) e uma produção inicial de onde se deve iniciar o processo de reconhecimento da linguagem.

Uma expressão matemática que envolva as operações de soma e subtração, por exemplo, pode ser reconhecida pela seguinte representação BNF (sendo *expr* a produção inicial):

1: *expr* → *expr* + *numero*

2: *expr* → *expr* − *numero*

3: *expr* → *numero*

No caso acima, assume-se que o *token numero* já é reconhecido pelo analisador léxico como uma sequência de dígitos. Pode-se assim, deduzir que 9 + 32 - 4 é uma *expr*, uma vez que (vide figura B.1.2):

1: 9 é uma *expr*, de acordo com a produção 3;

2: 9 + 32 é uma *expr*, de acordo com a produção 1;

3: 9 + 32 − 4 é uma *expr*, de acordo com a produção 2.

Durante o processo de reconhecimento do programa de entrada (em SDL) através da analise sintática, as estruturas do programa são armazenadas em memória em uma hierarquia lógica de forma a facilitar a próxima fase (geração de código).

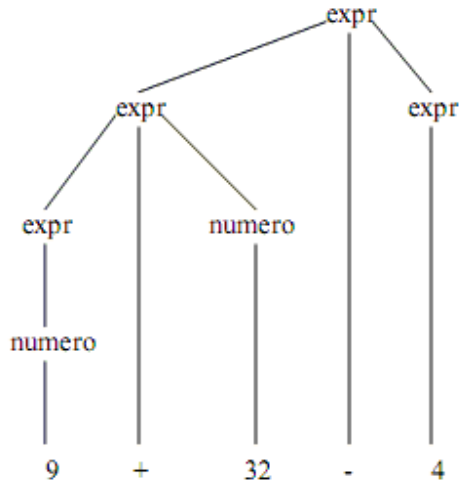


Figura B.2: Reconhecimento de uma expressão

B.1.3. Geração de código

A fase final do compilador é a geração de código, isto é, a transformação do programa reconhecido em alguma coisa executável (ou em um programa escrito na linguagem definida como destino). No caso do gerador de código desenvolvido neste trabalho, esta etapa é reponsável pela geração do código Java que representa o programa em SDL reconhecido pela ferramenta.

B.2. JavaCC

Neste contexto, o JavaCC (*Java Compiler Compiler*) se insere como uma ferramenta que auxilia na construção de um compilador. Através de uma sintaxe simples, o JavaCC constrói um compilador, dada a representação em notação BNF da linguagem a ser reconhecida.

Além disso, uma série de comandos em Java podem ser inseridos após o reconhecimento de cada terminal / não-terminal para que se possa tomar ações de acordo com o que for reconhecido. Desta forma, pode-se armazenar uma representação do programa reconhecido para que em uma etapa posterior se gere código Java a partir destas representações.

No JavaCC, algumas extensões à notação BNF podem ser utilizadas, são elas:

1. *(expr)+* – Indica que o não-terminal *expr* pode ocorrer uma ou mais vezes;
2. *(expr)** – Indica que o não-terminal *expr* pode ocorrer zero ou mais vezes;
3. *(expr)?* – Indica que o não-terminal pode ou não ocorrer
4. *(expr1 | expr2)* – Indica que pode ocorrer o não-terminal *expr1* ou *expr2*;

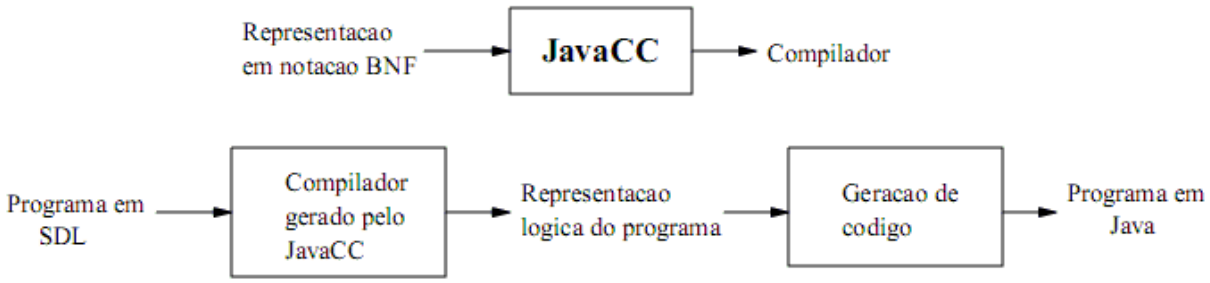


Figura B.3: Construção de um compilador com o uso de JavaCC

Para o reconhecimento dos programas em SDL foi convertida para o padrão JavaCC parte da gramática livre de contexto (apenas a gramática necessária para o desenvolvimento da rede proposta) descrita no documento Z.100 [14] do ITU-T que padroniza a linguagem SDL.

Apêndice C

Artigo Publicado

Anais do XXV Simpósio Brasileiro de Telecomunicações (SBrT'07) – Recife/PE – Set/2007

Proposta de uma Rede de Comunicação Automotiva Baseada no FlexRay

Daniel C. F. Rezende e Walter C. Borelli

***Resumo* – Este artigo apresenta a criação de um modelo executável de uma rede intraveicular baseada no protocolo de comunicação FlexRay. Sendo assim, é proposta neste artigo uma descrição do projeto e do modelamento em SDL de uma rede FlexRay, visto que isto não é encontrado na especificação padrão do FlexRay. Foram propostas algumas melhorias no protocolo para possibilitar a criação de uma completa e funcional rede FlexRay. Também são apresentadas simulações e sua completa validação. Além disso, são apresentados diagramas de troca de mensagens que descrevem o correto comportamento do FlexRay que podem ser usados como referência nos testes para validação de qualquer implementação usando este protocolo.**

***Palavras-Chave* – Protocolos de Comunicação, Sistemas Intraveiculares, FlexRay, SDL.**

***Abstract* – This paper presents the creation of an executable SDL model of an in-vehicle network based on the FlexRay protocol. Thus, it is presented a description of the design and the modelling in SDL of a functional FlexRay network, since this is not accomplished by the FlexRay Protocol Specification. Some improvements in the FlexRay specification were also proposed. Results of the system simulation and validation are also presented. Besides, message sequence charts are presented describing the correct behavior of FlexRay. Such diagrams may be used as reference in tests for validation of any application using this protocol.**

***Keywords* – Communications Protocols, FlexRay, In-Vehicle Systems, SDL.**

I. INTRODUÇÃO

Recentemente tem havido um aumento significativo na quantidade de eletrônica introduzida no carro, e é esperado que esta tendência continue enquanto as montadoras introduzirem avanços adicionais em segurança, confiabilidade e conforto [1].

Motivados pelas aplicações intraveiculares, várias companhias vêm investindo no projeto de controladores que pudessem gerenciar o tráfego de informações com interfaces através de um meio físico reduzido, geralmente um barramento serial, porém capaz de possibilitar a multiplexação dessas informações. A esta forma de conexão é dado o nome de Rede Intraveicular (In-Vehicle Networking).

A introdução de sistemas avançados de controle que combinam sensores múltiplos, atuadores e centrais eletrônicas, também chamadas de ECUs (Electronic Control Unit) está começando a demandar exigências na tecnologia de uma comunicação não encontrada atualmente nos protocolos de comunicação existentes.

As exigências adicionais para as aplicações futuras do controle intraveicular incluem combinação de taxas de dados mais elevadas, comportamento determinístico e sustentação da tolerância à falhas. Para comunicação não-crítica um número de protocolos tornou-se popular, tais como LIN (Local Interconnect Network), CAN (Controller Area Network) e MOST (Media Oriented Systems Transport). Para aplicações críticas (safety-critical) e mais complexas (aplicações X-by-Wire e aplicações relacionadas à transmissão e motor) o FlexRay surge como protocolo mais adequada a operações em tempo real [2].

A introdução desses sistemas X-by-Wire, que essencialmente significa substituir componentes mecânicos e hidráulicos por componentes eletrônicos conectados por fios sem a necessidade de componentes mecânicos sobressalentes, será um passo significante para a indústria automobilística [3]. Sistemas de controle de tração e estabilidade (Steer-by-Wire), de freios (Brake-by-Wire) e direção (Drive-by-Wire) são exemplos de sistemas X-by-Wire.

Este artigo apresenta uma proposta de um modelo SDL executável para uma rede de comunicação para aplicação automotiva baseada no protocolo FlexRay. A rede foi modelada usando a linguagem SDL (Specification and Description Language). O propósito deste trabalho é criar um modelo executável de uma rede FlexRay e a partir do qual produzir diagramas de trocas de mensagens (MSC - Message Sequence Charts) que possam ser utilizados para validar aplicações deste protocolo. Tais diagramas são de suma importância na geração de pacotes de testes (Test Suites) para aplicações específicas. Este artigo descreve o projeto e o modelamento em SDL de uma rede funcional FlexRay, visto que isto não é realizado na especificação padrão do FlexRay [5] que apenas apresenta o comportamento dos mecanismos principais do protocolo. Algumas melhorias na especificação do FlexRay foram propostas para a elaboração desta rede usando SDL.

Este artigo é organizado como segue: Seção II descreve o protocolo de comunicação FlexRay, Seção III descreve o modelo do sistema e também apresenta as melhorias feitas na especificação do protocolo. Seção IV apresenta alguns resultados da simulação do sistema, sua validação e a geração dos diagramas de trocas de mensagens. Conclusões são apresentadas na Seção V.

II. PROTOCOLO FLEXRAY

O protocolo FlexRay surgiu como resultado da cooperação entre BMW e DaimlerChrysler depois que as duas montadoras automotivas perceberam que as soluções atuais não satisfariam suas necessidades para aplicações futuras incluindo X-By-Wire. Como resposta a isso, o Consórcio FlexRay [1] foi formado com o objetivo de desenvolver um novo protocolo, chamado FlexRay [4] e [5]. Este novo protocolo deveria ser a solução não só para a introdução dos sistemas X-By-Wire, mas também para a substituição de

Daniel Cesar Felisberto Rezende e Walter da Cunha Borelli, Departamento de Telemática, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas-SP, E-mails: {danielre, borelli}@dt.fee.unicamp.br.

alguns protocolos adotados atualmente, assim reduzindo o número total de redes intraveiculares [6].

FlexRay é uma arquitetura eletrônica aberta, comum e escalável para aplicações automotivas. Este sistema pode ser operado no modo de canal único ou no modo com dois canais, fornecendo, assim, redundância quando necessário. O FlexRay permite transmissão de dados síncronos e de dados assíncronos. Através da transmissão síncrona, outros nós na rede recebem mensagens orientadas por tempo (time-triggered) em um tempo de latência predefinido; através da transmissão assíncrona, as mensagens chegam a seus destinos de forma mais rápida ou mais devagar, dependendo da prioridade atribuída a elas (event-triggered). Atualmente, o sistema FlexRay pode alcançar taxas de até 10 Mbps.

É esperado que o FlexRay seja o sistema de comunicação padrão para aplicações automotivas de controle interconectando ECUs em sistemas automotivos futuros de alta-velocidade [6].

III. Modelo do Sistema

Para criar uma rede FlexRay completa e funcional sua estrutura teve que ser projetada. Dessa forma, a arquitetura do sistema, a topologia da rede, a integração entre os principais mecanismos dentro do controlador de comunicação e a interface que conecta o controlador de comunicação ao canal de comunicação foram todos criados e descritos usando a linguagem SDL.

É considerado neste trabalho um sistema com três nós interconectados através da topologia de barramento. O sistema de barramento FlexRay modelado juntamente com o esboço da arquitetura do nó é ilustrado na Fig. 1.

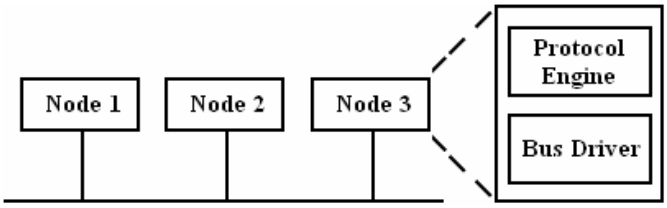


Fig. 1. Exemplo de sistema FlexRay e arquitetura do nó.

O sistema é uma entidade representada por um quadro que delimita a rede modelada. O que está fora do escopo deste quadro é chamado de ambiente e não está especificado em SDL. Fig. 2 ilustra o sistema da rede FlexRay modelada contendo quatro blocos. Esses blocos descrevem um conjunto de nós interconectados por um barramento. Os blocos chamados *Node1*, *Node2* e *Node3* representam os nós na rede. O bloco *Bus* representa o meio físico.

Cada bloco que representa um nó é conectado a cinco canais. Esses canais permitirão ao sistema modelado interagir com outras partes não especificadas neste modelo, que são: o CHI (interface entre o host e o controlador de comunicação) e alguns dos mecanismos do núcleo do protocolo FlexRay (codificação/decodificação e sincronismo do clock). O processo de codificação e decodificação é representado pelo termo CODEC. O mecanismo de sincronismo do clock é dividido em três processos: o processo de inicialização do sincronismo do clock (CSS), o processo de sincronismo do clock (CSP) e o processo de geração de macrotick (MTG). Os sinais que trafegam em cada direção através dos canais são indicados por listas de sinais anexadas às extremidades dos canais. A comunicação entre os nós da rede é alcançada codificando a carga útil de um pacote de dados (enviada pelo CHI) em frames ou símbolos e transmitindo através do barramento.

As descrições do comportamento dentro do Nó1 e do Nó2 são as mesmas. Somente o Nó3 tem descrição de comportamento diferente, pois ele foi adaptado para não ser um nó coldstart (ver seção IV).

Fig. 3 ilustra como o nó foi descrito em SDL. Cada bloco Nó contém um bloco controlador de comunicação (*Communication Controll*er) e um bloco controlador de barramento (*Bus Driver*). Nesta arquitetura o CHI é usado como interface de controle e de dados entre o sistema host e o mecanismo do protocolo FlexRay.

O bloco *Communication Controller* recebe e envia sinais para o CHI, para o bloco *Bus Driver* e para os outros componentes do mecanismo do protocolo FlexRay. O bloco *Bus Driver* troca sinais diretamente com o bloco *Communication Controller* e indiretamente com o CHI, o processo CSS e com o barramento através do ambiente.

Três dos principais mecanismos que compõem o protocolo FlexRay estão descritos como processos dentro do bloco *Communication Controller*. Fig. 4 ilustra os três processos no bloco *Communication Controller*.

A finalidade do processo *POC* (Protocol Operation Control) é reagir aos comandos do host e às condições do protocolo ativando mudanças coerentes nos mecanismos de maneira síncrona, e fornecer ao host o estado atual dos mecanismos em relação às mudanças.

O processo *MAC* (Media Access Control) define o controle de acesso ao meio físico (barramento) através do gerenciamento do ciclo de comunicação recorrente.

O processo *FSP* (Frame and Symbol Processing) verifica a correta temporização dos frames e símbolos em relação ao esquema TDMA, aplica testes sintáticos adicionais nos frames recebidos e verifica a correção semântica dos frames recebidos.

A especificação do FlexRay não descreve o controlador de barramento (*Bus Driver*) usando SDL, logo este teve que ser criado. O controlador de barramento é um componente eletrônico que consiste de um transmissor e um receptor que conecta o controlador de comunicação a um canal de comunicação.

O bloco *Bus Driver* é ilustrado na Fig. 5 e contém dois processos: *Transmitter* e *Receiver*. Para simplificar o sistema FlexRay, mas ainda seguindo as regras de operação do protocolo, os processos Transmissor e Receptor também foram projetados para realizar parte do trabalho do processo CODEC. Por exemplo, quando o sinal *transmit_frame_on_A (vType, vTF*) chega ao bloco *Bus Driver*, ele é guiado através da rota de sinal (signal route) *FromCC* para o processo *Transmitter* e depois ele é convertido para o sinal *Frame vType, vTF)* e finalmente enviado para o barramento.

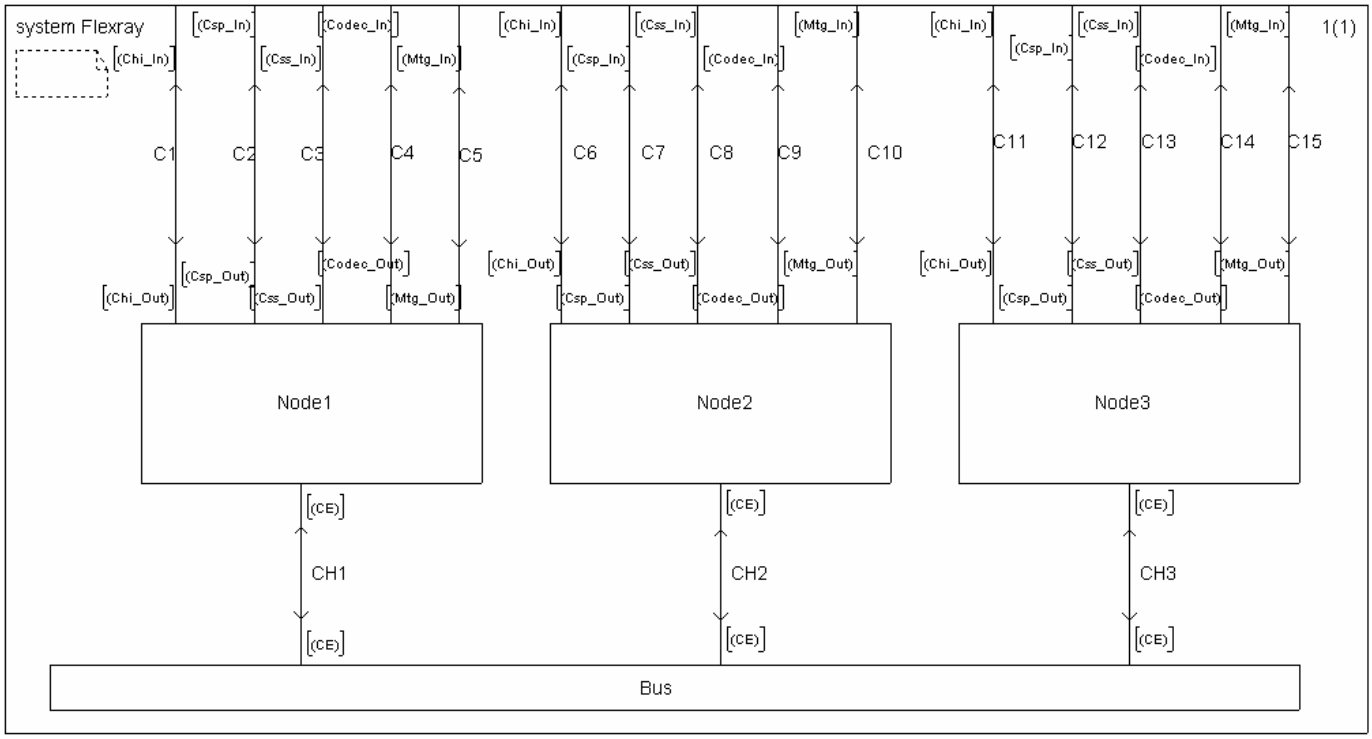


Fig. 2. Nível de sistema do modelo SDL.

No processo Receiver ocorre a conversão inversa. Quando o sinal *Frame (vType, vTF)* chega, seu parâmetro *vTF* (variável que contém os dados do frame a ser transmitido) é atribuído ao parâmetro *vRF* (variável que contém os dados do frame recebido) e um sinal *frame_decoded_on_A (vRF)* é enviado. O parâmetro *vType* (variável que contém o tipo de transmissão) é usado para diferenciar a recepção do elemento de comunicação.

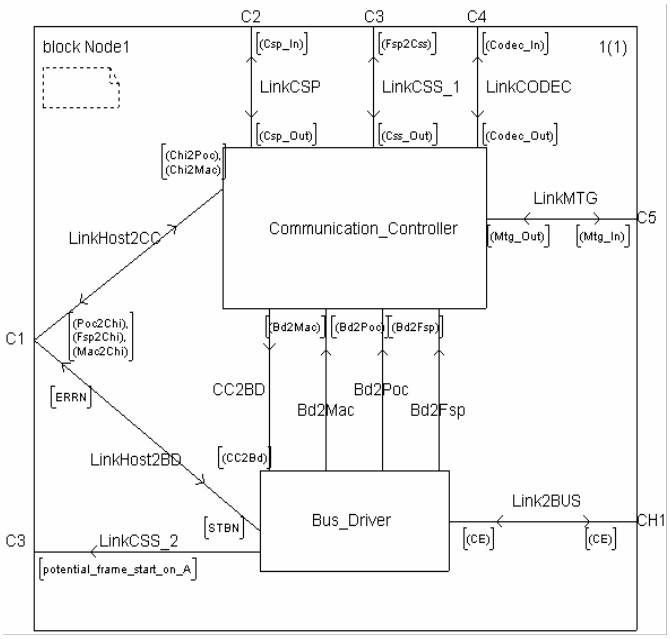


Fig. 3. Descrição do nó.

Uma interface para o CHI também foi especificada no bloco *Bus Driver* e consiste de um sinal *STBN* (Standby Not) que é usado para controlar o modo de operação do controlador do barramento e um sinal *ERRN* (Error Not) que é usado para indicar erros detectados.

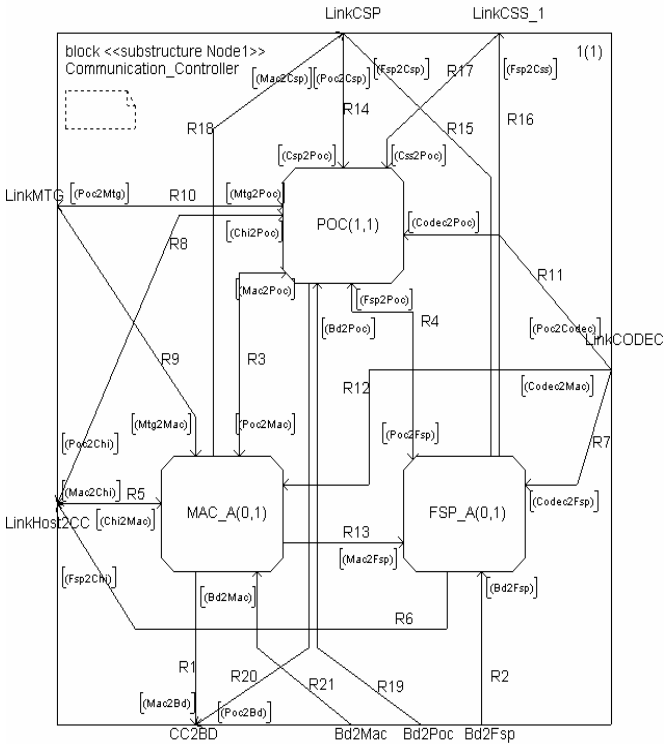


Fig. 4. Descrição do núcleo do protocolo.

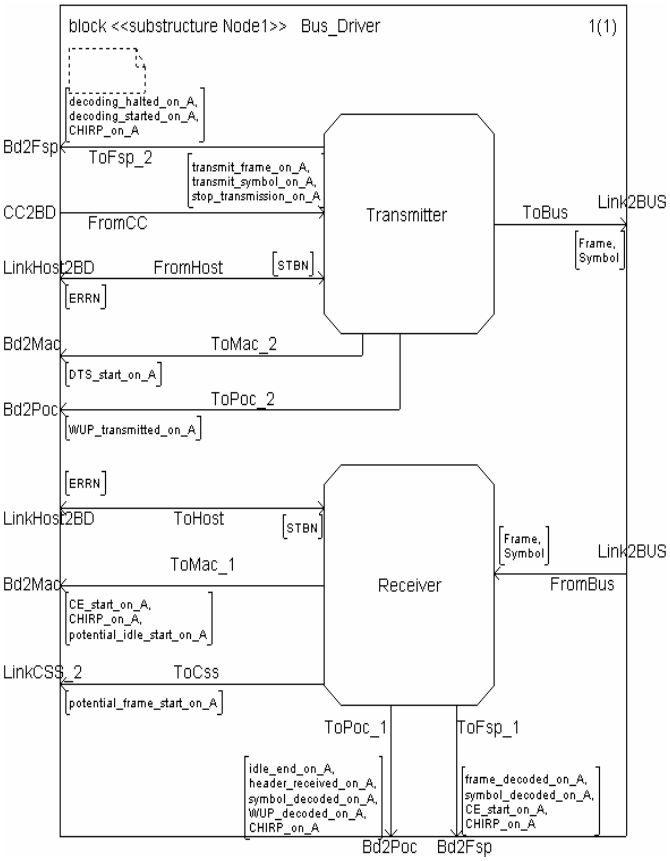


Fig. 5. Descrição do controlador de barramento.

Além da descrição completa da estrutura da rede, algumas melhorias e modificações também foram feitas em relação à especificação do FlexRay a fim de gerar um sistema executável usando SDL.

As principais alterações foram:

- Substituição de todas as construções de macros descritas em [5] por construções de procedimentos.
- Criação do bloco *Bus Driver*.
- Adaptação dos tipos existentes de temporizadores do FlexRay para um único tipo de temporizador baseado na unidade microtick.
- Substituição de todas as construções task contendo textos informais relacionados ao CHI por sinais de saída usando as construções de output.
- Substituição de alguns termos descritos em [5] por novos termos devido a conflito com palavras-chave utilizadas pela ferramenta SDL.
- Criação de um procedimento *CycleCounter* para verificar o ciclo de comunicação atual.
- Substituição da construção task *import_vTCHI_from_the_CHI* no procedimento *ASSEMBLE_STATIC_FRAME_A* por um sinal de entrada recebido do CHI.

Alguns mecanismos do protocolo especificados em [5] usam construções de macros com o exclusivo propósito de simplificar a apresentação do protocolo em SDL. Porém, esta prática não é aconselhável ao implementar sistemas reais portáteis. Macros são geralmente dependentes de certas ferramentas SDL, logo o projetista poderia ter dificuldade ao tentar compilar seu modelo utilizando diferentes editores de SDL. Dessa forma, todas as construções de macros foram substituídas por construções de procedimentos.

A criação do bloco *Bus Driver*, como explicado anteriormente, foi fundamental na geração de um modelo SDL executável, já que este componente não é especificado em [5].

A representação do tempo no protocolo FlexRay é baseada em uma hierarquia que inclui microticks, macroticks e sampleticks. Vários mecanismos do FlexRay necessitam de temporizadores que medem certo número de microticks ou macroticks. A especificação do FlexRay usa uma extensão dos temporizadores do SDL para realizar tal função, porém esta extensão não é suportada pela ferramenta SDT utilizada (TAU SDL Design Tool v.4.2).

Como existe uma relação definida entre a unidade de tempo microtick (μT), a unidade de tempo macrotick (MT) e a unidade de tempo da amostragem de bits sampleticks (ST), o temporizador microtick foi definido como sendo o único tipo de temporizador presente em todo o sistema. Logo, todos os outros temporizadores (macrotick e sampletick) tiveram seus valores recalculados como segue: 1MT é igual a 40μT e 1ST é igual a 1μT. Assim, os temporizadores sampletick foram diretamente convertidos para temporizadores microtick e os temporizadores macrotick tiveram seus valores multiplicados por 40, valor representado pela constante *pMicroPerMacroNom*. Por exemplo, o temporizador *tMinislot* que tem valor igual a 40MT (representado pela constante *gdMinislot*) é declarado como sendo multiplicado pela constante *pMicroPerMacroNom* (número inteiro de microticks por cada macrotick). Fig. 6 ilustra a declaração dos temporizadores no processo MAC_A.

```
timer tSlotBoundary := gdStaticSlot * pMicroPerMacroNom;
timer tMinislot := gdMinislot * pMicroPerMacroNom;
```

Fig. 6. Declaração dos temporizadores.

Alguns termos descritos em [5] entraram em conflito com as palavras-chave da ferramenta SDL, por exemplo, *ALL*, *STATE*, *CHANNEL* e *ACTIVE*. A solução encontrada foi substituir esses termos por outras palavras livres de conflito. Os termos foram substituídos, respectivamente, por *TODOS*, *ESTADO*, *CANAL* e *ATIVO*.

Para gerar um modelo SDL mais próximo de uma implementação real algumas construções task, contendo textos informais relacionados à exportação de variáveis do CHI, foram substituídas por sinais de saída usando as construções output. O mesmo acontece no procedimento *ASSEMBLE_STATIC_FRAME_A* localizado no processo *MAC_A*, mas ao contrário do exemplo anterior, uma construção input de sinal de entrada está substituindo a construção task original *import_vTCHI_from_the_CHI*. Então agora, o sistema pode receber dados do host pelo CHI.

As decisões de mudança dos modos nos mecanismos feita pelo processo *POC* no final de cada ciclo depende se o número atual do ciclo é par ou ímpar, porém [5] não descreve nenhum mecanismo para verificar o estado do ciclo. Assim, foi criado um procedimento chamado *CycleCounter* para verificar o ciclo de comunicação atual durante a operação do FlexRay.

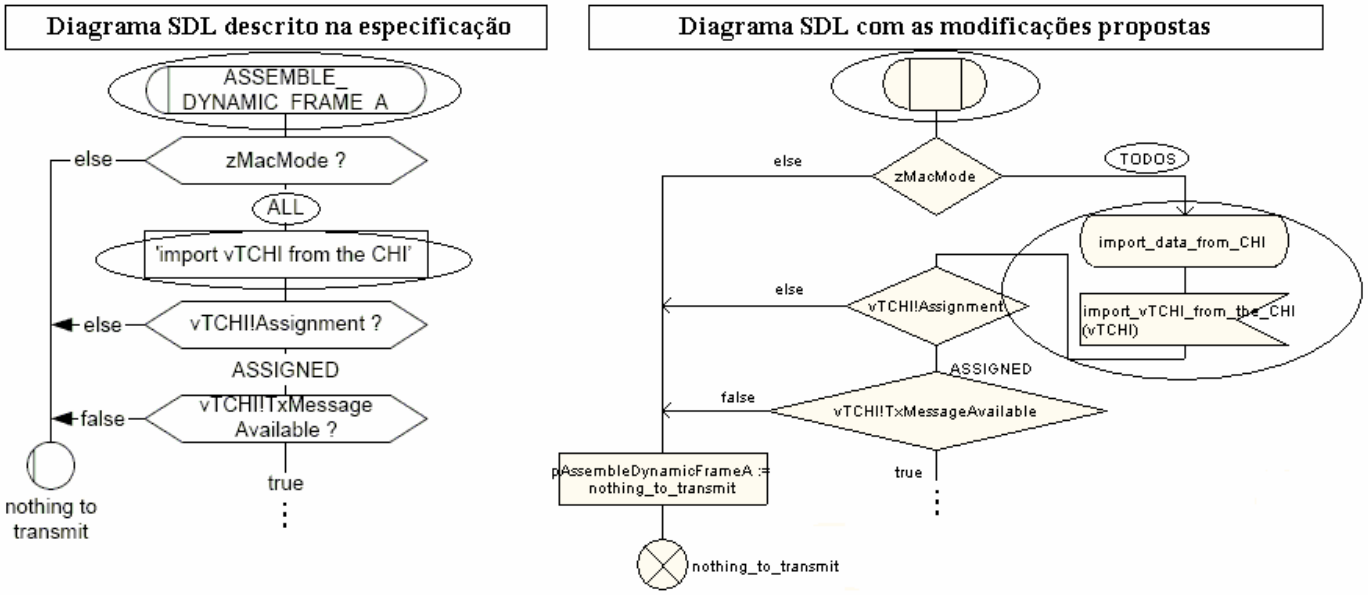


Fig. 7. Do lado esquerdo, o diagrama SDL obtido em [5]. Do lado direito, o diagrama SDL proposto. Os círculos destacam onde as alterações ocorreram.

Na Fig. 7 estão ilustrados três das modificações mencionadas acima que foram feitas em [5] a fim de aprimorar o modelo SDL, que são: a descrição do diagrama usando procedimento ao invés de macro, a substituição do termo *ALL* por *TODOS* e a substituição da construção task *import_vTCHI_from_the_CHI* pelo sinal de entrada *import_vTCHI_from_the_CHI (vTCHI)* através da construção input, onde *vTCHI* é a variável que armazena os dados enviados pelo host.

IV. Simulação, Validação e Geração de MSCs da Rede FlexRay

Depois de especificar a rede FlexRay proposta, as funcionalidades desejadas foram completamente testadas. O sistema foi integralmente validado através da ferramenta Validator disponível no pacote SDT da Telelogic.

Nesta fase foi possível detectar alguns erros dinâmicos (erros gerados em tempo de execução) como, por exemplo, o consumo implícito de sinal (implicit signal consumption) no procedimento *Wakeup*. Este problema aconteceu porque foram ativados dois temporizadores e quando um deles expirou, seu sinal foi consumido, ao passo que o sinal proveniente do outro temporizador foi descartado. Tal erro foi facilmente descoberto depois da execução da ferramenta Validator, notando-se que não havia um comando para desativar o segundo temporizador depois do término do primeiro. A solução foi adicionar um comando para realizar tal função (Fig. 8).

Depois da correção de todos os erros dinâmicos, o sistema foi novamente validado e então todos os símbolos esperados foram alcançados. Fig. 9 mostra o gráfico de cobertura dos símbolos gerado pela ferramenta de validação do SDL.

Após a completa validação do sistema, foram simuladas as principais funcionalidades do sistema FlexRay para a garantia da validade do sistema. Para tanto foi utilizada a ferramenta Simulator, que também é parte integrante do pacote SDT. Esta ferramenta permite o acompanhamento total da simulação gerando diagramas de trocas de mensagens (MSC - Message Sequence Charts), que permitem visualizar a interação entre os diversos componentes do sistema.

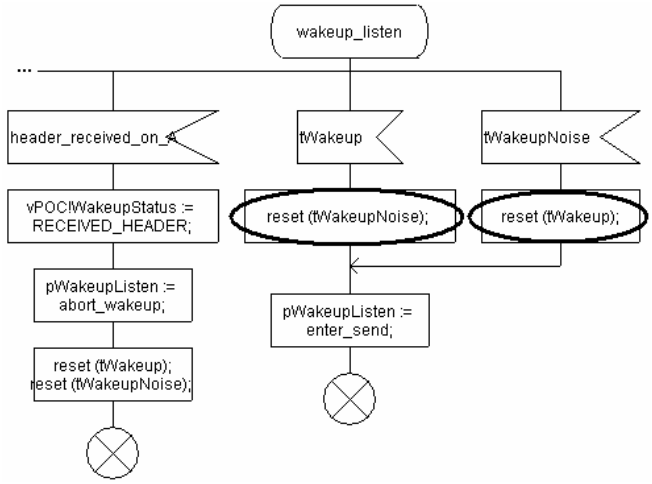


Fig. 8. O comando reset foi adicionado para evitar sinal implícito.

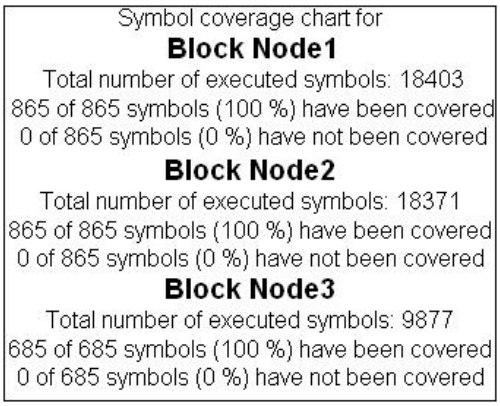


Fig. 9. Cobertura de símbolos dos Nós 1, 2 e 3.

Para a simulação do modelo FlexRay proposto foram utilizados os cenários para o início de comunicação da rede

descritos em [5]. O ato de começar o processo de inicialização da rede é chamado de coldstart. Somente alguns nós são permitidos inicializar a rede, chamados nós coldstart. De acordo com a especificação, três diferentes tipos de caminhos podem ser seguidos pelo nó dependendo de sua configuração. O primeiro caminho refere-se ao nó coldstart pioneiro na inicialização; o segundo caminho refere-se aos nós coldstart que não foram os pioneiros na inicialização e o terceiro caminho refere-se aos nós que não são coldstart. A partir desta descrição foram gerados nove diagramas de trocas de mensagens (MSC) mostrando detalhadamente o correto comportamento do mecanismo do protocolo dentro de cada tipo de nó. Para cada tipo de caminho, três cenários diferentes foram simulados, dependendo da verificação de erro executada ao final de cada ciclo de comunicação. Dessa forma todas as situações possíveis que ocorrem durante a inicialização e a operação normal de uma rede FlexRay foram simuladas e descritas em forma de diagramas. Tais diagramas podem ser utilizados na criação de pacotes de testes para implementação de qualquer aplicação usando o FlexRay. Estes pacotes de testes podem ser produzidos por ferramentas específicas (TTCN) e estão também disponíveis no software SDT usado neste trabalho.

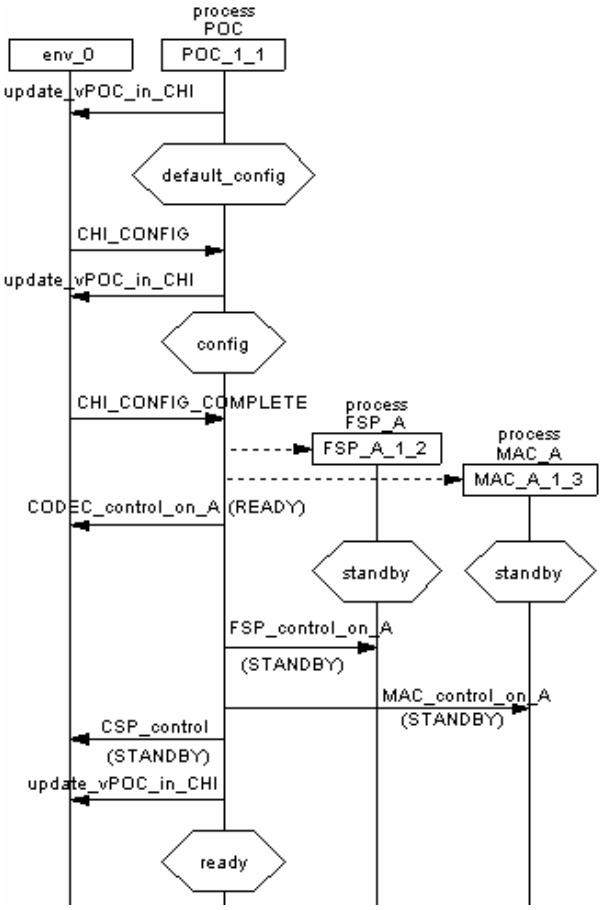


Fig. 10. MSC que descreve o processo de inicialização da rede FlexRay.

A Fig. 10 ilustra o início de um dos diagramas gerados. Esse diagrama refere-se ao cenário de um nó coldstart pioneiro na inicialização da rede. Antes de entrar no estado *default_config* o processo *POC* inicializa o valor de suas variáveis. Após isto feito, ele aguarda um sinal do host para entrar no estado *config* e assim habilitar a configuração da rede. Ao receber o sinal *CHI_CONFIG_COMPLETE* o processo *POC* cria todos os outros processos principais do protocolo, passa para o estado *ready* e aguarda a fase de inicialização da rede.

V. CONCLUSÃO

O trabalho apresentado neste artigo propôs a geração de um modelo SDL executável de uma rede intraveicular baseada no protocolo de comunicação FlexRay. Para criar uma completa e funcional rede FlexRay sua estrutura teve que ser projetada. Dessa forma, a arquitetura do sistema, a topologia da rede, a integração entre os principais mecanismos dentro do controlador de comunicação e a interface que conecta o controlador de comunicação ao canal de comunicação foram todos criados e descritos usando a linguagem SDL. O sistema foi modelado usando a especificação do protocolo FlexRay para fornecer um modelo executável dos mecanismos do protocolo e a partir do qual produzir diagramas de trocas de mensagens que possam ser utilizados para validar aplicações deste protocolo. Tais diagramas são de suma importância na geração de pacotes de testes (Test Suites) para aplicações específicas. Algumas melhorias na especificação do FlexRay foram propostas para a elaboração desta rede usando SDL.

Para a concretização deste trabalho fez-se uso do conjunto de ferramentas SDT, utilizando-se o Simulator para a simulação dos casos de uso mais importantes do sistema FlexRay e o Validator para a validação completa do sistema. Utilizando-se de tais ferramentas, todas as possíveis situações puderam ser testadas, o que garante a validade do modelo proposto. Pode-se desta forma, facilitar o processo de uma eventual implementação, uma vez que a lógica do sistema já está verificada e não possui erros.

REFERÊNCIAS

[1] FlexRay Consortium. FlexRay. Available at: <http://www.flexray.com> Access: Jul. 2006.

[2] E. Armengaud et al. “A monitoring concept for an automotive distributed network - the FlexRay example”. In: 7th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2004), 2004.

[3] G. Leen, D. Hefferman, “In-vehicle networks, expanding automotive electronic systems”. In: IEEE Transaction on Computers, Jan. 2002, pp. 88-93.

[4] J. Berwanger et al. “FlexRay – the communication system for advanced automotive control systems”. In: SAE 2001 World Congress, Society of Automotive Engineers, Detroit, MI, Apr. 2001.

[5] FlexRay Consortium. FlexRay Communications System - Protocol Specification Version 2.1 Revision A. Dec. 2005.

[6] T. Nolte, H. Hansson, L. Bello, “Implementing next generation automotive communications”. In: Proceedings of the 1st Embedded Real-Time Systems Implementation Workshop (ERTSI'04) in conjunction with the 25th IEEE International Real-Time Systems Symposium (RTSS'04), Lisbon, Portugal, Dec. 2004.

[7] Telelogic AB. Telelogic TAU 4.2 SDL Suite Getting Started: Technical report, Telelogic AB Sweden, Sep. 2001.

[8] R. Braek, “SDL basics”. In: Computer Networks and ISDN Systems, Jun. 1996, pp. 1585-1602.

[9] L. Doldi, Validation of Communications Systems with SDL. Chichester, Hoboken (NJ), Wiley, 2003.